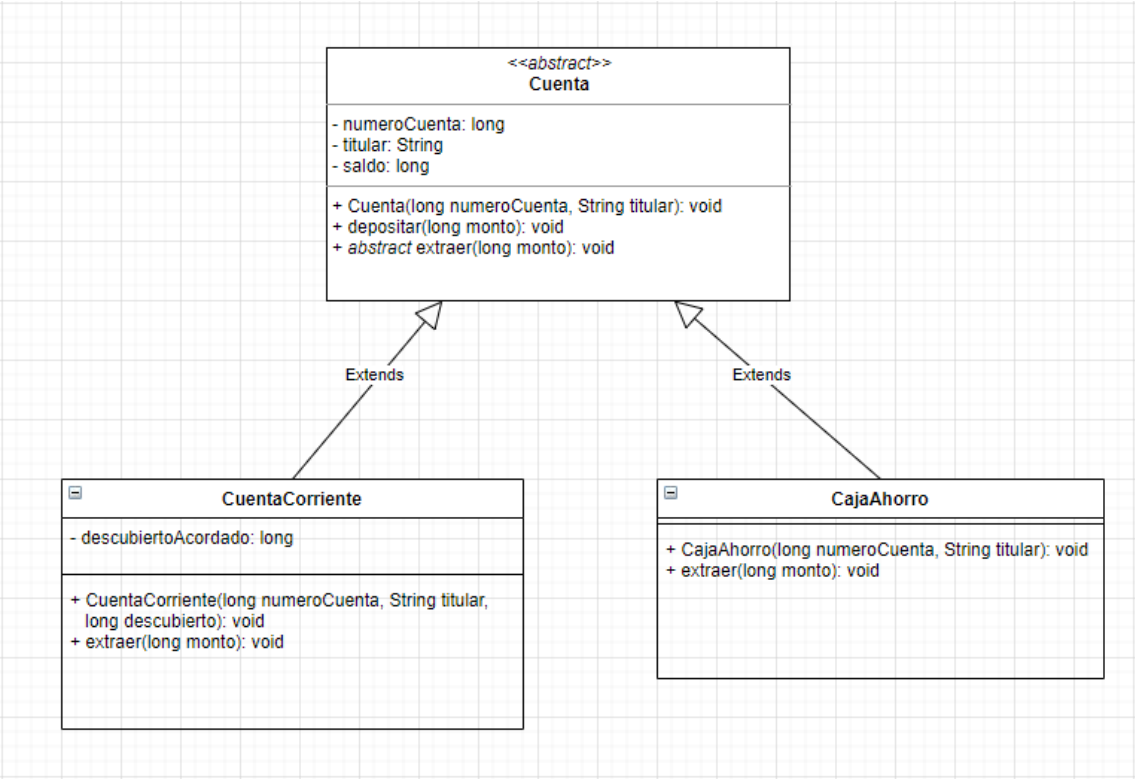
ATIX LABS _

EVALUACIÓN Developer SR _ Semi SR

1. Resolución de problemas de programación y diseño

A) Tome el código que sigue – que representa una cuenta bancaria – y responda las siguientes preguntas:

- ¿Le parece que está bien diseñada la clase? Si no lo está, pero funciona, ¿la cambiaría o agregaría clases nuevas?
  - No me parece que esté bien diseñada la clase, aunque a simple vista pareciera compilar y funcionar. Sí, le agregaría una jerarquía de clases polimórficas, poniendo como clase base abstracta a ***Cuenta*** y que las demás clases concretas hereden de ésta y extiendan su comportamiento específico. En este caso sería **CuentaCorriente** y **CajaAhorro**.
- Si cambiara el diseño, exponga todo lo que cambiaría con un diagrama de clases de UML. Si algún método debe cambiarlo mucho, escríbalo.
  - Link a diagrama UML



- ¿Es seguro hacer estos cambios? ¿Por qué? ¿Qué precauciones tomaría?
  - Es seguro si se detectan todos los puntos de impacto que trae este refactor en el proyecto. Tendría tests unitarios con coverage alto como precaución, para luego correrlos y verificar si alguno falla, como primera instancia.

- ¿Agregaría getters y setters? ¿Cuáles? ¿Por qué?
  - Sí, todos getters y setters públicos salvo el setter del saldo que es *protected* para la jerarquía de clases de ***Cuenta***. Agregué estos getters y setters porque el código base de la clase ***Cuenta*** tenía todos sus atributos como *private.*
- ¿Su solución usa algún patrón de diseño? ¿Cuál?
  - No necesité, solo utilicé herencia.

2. Aspectos conceptuales

A) Explique el uso del patrón Strategy. Una vez explicado, conteste: ¿Cuántas instancias necesita de cada clase de estrategia? ¿Hay algún otro patrón que lo ayude en esto? Si lo hay, muestre un pequeño ejemplo en código.

- El patrón Strategy es un tipo de patrón de diseño de comportamiento que ayuda en el desarrollo de software aportando desacoplamiento en el código, separando responsabilidades y resultando en un código más legible y ordenado, evitando If's anidados o switchs muy grandes. Esto permite realizar refactors más fácilmente en el futuro.
  Básicamente se necesitan tres actores: el **Contexto** que contiene la estrategia concreta, la **Estrategia** propiamente dicha que es una interfaz común para todos los tipos distintos de implementaciones y la **EstrategiaConcreta** que implementa el método declarado en la interfaz.
  Y su uso es simplemente la invocación de un método en el **Contexto** que internamente tiene seteado una **EstrategiaConcreta** que ejecuta su propia implementación de algún algoritmo.

  Las instancias que necesita cada clase de estrategia creo que dependerá del contexto en el que se esté desarrollando, pero como la mayoría de los backends son web services, crearía una instancia nueva por cada request que esté procesando en el momento. Un patrón que veo que ayudaría a esto sería el patrón Factory. Un ejemplo podría ser el siguiente:

```
public class StrategiesFactory {

    private static Map<String, Strategy> strategies = new Hashmap<>();

    static {
        strategies.put("ConcreteStrategyA", new ConcreteStrategyA());
        strategies.put("ConcreteStrategyB", new ConcreteStrategyB());
    }

    public static Strategy getInstance(String type) {
        return strategies.get(type);
    }
}
```

B) Enumere todas las ventajas que conozca de escribir pruebas unitarias automatizadas antes de escribir el código funcional.

- Básicamente poder implementar Test Driven Development como metodología de desarrollo con todos sus beneficios.
- Los tests forman parte de la documentación del código.
- Se evita escribir código de más y se desarrolla específicamente la funcionalidad requerida (KISS).
- Cada clase tiene agrupada en una Test Class sus respectivos tests sobre su funcionalidad.
- Se logra un desarrollo evolutivo, se empieza por las funcionalidades mas simples hasta llegar a casos mas complejos que se complementan con funcionalidades previamente ya testeadas.
- Test coverage y refactorizaciones más simples.

C) ¿Cuándo utiliza el patrón Observador? ¿Qué ventajas tiene?

- Un buen momento para utilizar el patrón Observador es cuando varios objetos dependen de los cambios de estado de otro objeto al que se suscriben, para luego ser notificados de estos cambios.

  La ventaja principal que veo es que usando este patrón se evita recurrir a malas prácticas de programación como por ejemplo 'Busy Waiting' que puede resultar muy poco performante, porque al suscribirnos a otro objeto, el encargado de avisarnos es él y no nosotros.

3. Bases de datos y SQL

A) Devuelva los usuarios cuyos nombres de persona empiecen por "Jorg"

```
SELECT * FROM Usuario as u
INNER JOIN Persona as p ON u.Id = p.idUsuario
WHERE p.FirstName LIKE 'Jorg%'
```

B) Devuelva los meses en los cuales la cantidad de usuarios que cumplen años es mayor a 10.

```
SELECT MONTH(fechaNac)
FROM Persona
GROUP BY MONTH(fechaNac)
HAVING COUNT (MONTH(fechaNac)) > 10
```