Cada ejercicio del examen se va a calificar con un valor entre 0 y 1. Para aprobar el examen se deberán cumplir todas las siguientes condiciones:

- En cada uno de los dos primeros ejercicios se debe tener al menos un puntaje de 0,6.
- Se debe sumar como mínimo el 60% del puntaje total obtenido mediante la ponderación de cada ejercicio (ver el coeficiente de ponderación en cada ejercicio).

1 - Ejercicio de modelado (se recomienda leer todo el ejercicio antes de comenzar). Se desea modelar parte de un sistema mediante el paradigma de objetos.
Ponderación: 60%

Contexto:
Nos asignaron el modelado de una parte de un sistema de asignación de descuentos en las tarifas de transporte público. Se necesita conocer el total de plata que gasta un usuario al pagar un viaje utilizando tarjetas tipo SUBE.

Requisitos:
Un usuario paga el viaje realizado utilizando una de sus múltiples tarjetas (tipo SUBE). El precio a pagar por el viaje depende de la distancia recorrida y del tipo de tarjeta que utilice el usuario. A continuación se describen las tarjetas y sus características relevantes:

Tarjeta tipo **Trabajador**:

- Cuando se utiliza, el precio del viaje es 150 * distancia recorrida.

Tarjeta tipo **Exento**:

- Cuando se utiliza, entre el día 16 y el fin de mes, el precio del viaje se calcula como se calcula con el tipo de tarjeta *Trabajador*.
- Cuando se utiliza, entre el día 1 y el 15 del mes, el precio del viaje es de 25 * distancia recorrida.

Se pide:

- Modelar en UML (diagrama de secuencia, con objetos y mensajes) el caso completo para los siguientes escenarios:

  A) Se necesita conocer el total pagado por un usuario que tiene dos tarjetas del tipo *Trabajador* y realiza los siguientes viajes:
     - Un viaje el día 2024/09/01 de 35 km utilizando la tarjeta con ID "id1".
     - Un viaje el día 2024/09/03 de 15 km utilizando la tarjeta con ID "id2".

  B) Se necesita conocer el total pagado por un usuario que tiene una tarjeta de tipo Exento ("id1") y una de tipo *Trabajador* ("id2") y realiza los siguientes viajes:
     - Un viaje el día 2024/09/01 de 10 km utilizando la tarjeta con ID "id1"
     - Un viaje el día 2024/09/20 de 20 km utilizando la tarjeta con ID "id1"
     - Un viaje el día 2024/09/22 de 20 km utilizando la tarjeta con ID "id2"

- Modelar en UML (diagrama de clases) que utilizó en los diagramas de secuencias de los casos anteriores. Use nombres adecuados para todas las clases, métodos y asociaciones que defina. Incluya, al menos, todos los métodos que utilizo en los diagramas de secuencia.

IMPORTANTE

En cada diagrama de secuencia mostrar la inicialización de los objetos involucrados

IMPORTANTE

Mantenga el nivel de abstracción que permita extender el modelo cumpliendo los principios de diseño que conoce.

2 - Ejercicio conceptual
Ponderación: 25%

Diagrama 1

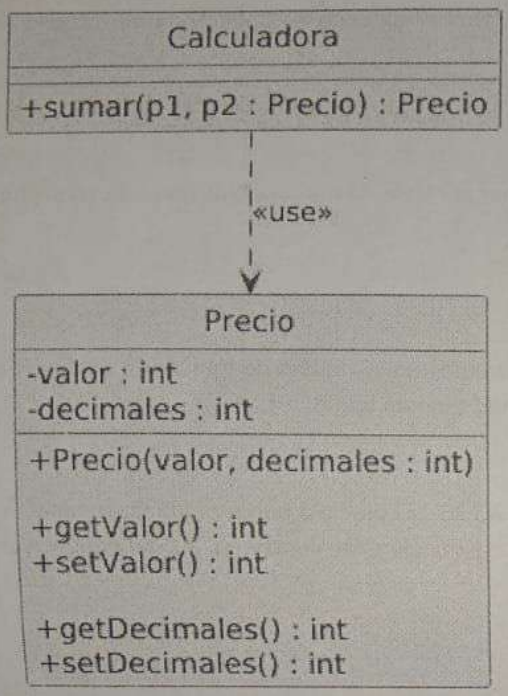
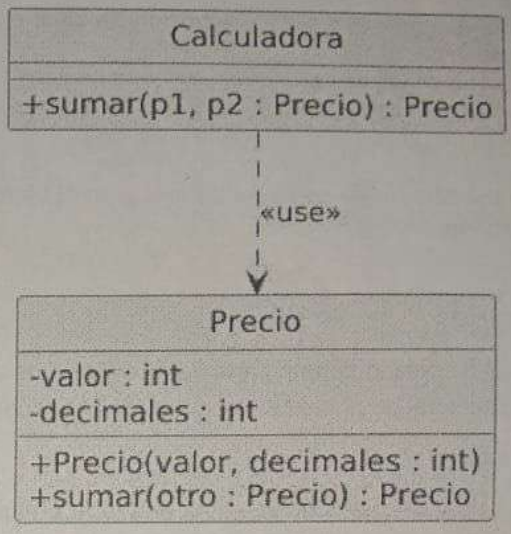


Diagrama 2



A. Cuales de las soluciones elegiría y por que motivo dando un ejemplo **utilizando las clases del enunciado**?
B. Cuales principios de diseño y/o pilares del paradigma podría comprometer la solución con más problemas? Para cada problema debe brindar un ejemplo **utilizando las clases del enunciado**.

---

3 - Ejercicio de lecturas obligatorias (contestar sobre esta hoja, no es necesario justificar)
Ponderación: 15%

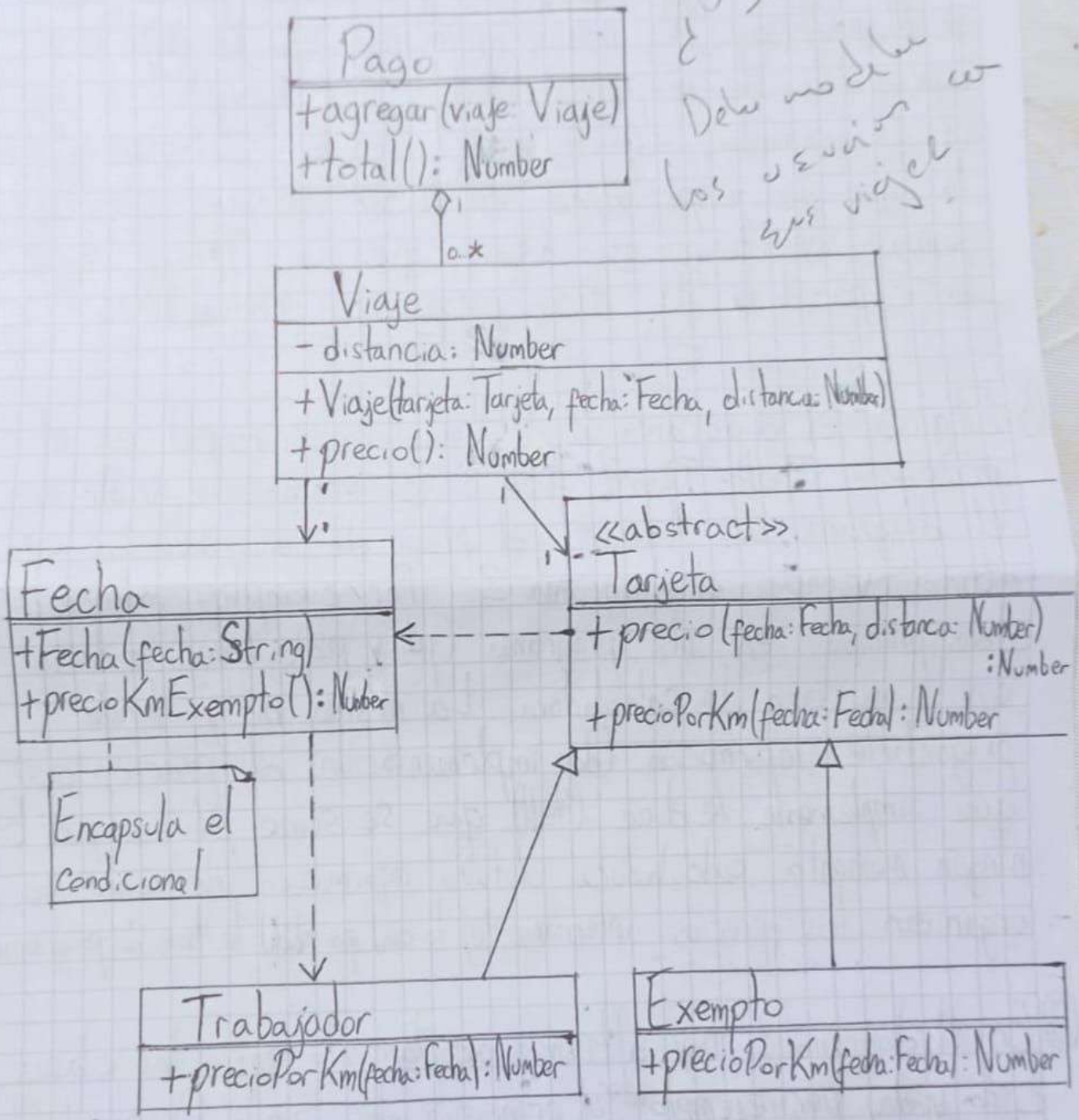
"Design Principles Behind Smalltalk", Dan Ingalls

- ☐ A) Un lenguaje debe diseñarse en torno a una poderosa *metáfora* que pueda aplicarse uniformemente en todas las áreas.
- ☐ B) Cuando un sistema está realmente bien construido, tanto los usuarios como los implementadores disponen de un gran *apalancamiento*.
- ☐ C) Ningún componente de un sistema *complejo* debe depender de los detalles internos de cualquier otro componente.
- ☑ D) Opciones A, B y C son correctas.
- ☐ E) Opciones A y C son correctas.

Según el artículo "Unit Testing Guidelines", Petroware

- ☐ A) No define que significa unidad de prueba (unit test).
- ☑ B) La unidad de prueba tiene que ser pequeña y ejecutarse rápidamente.
- ☑ C) Las unidades de pruebas tienen que ser independientes entre sí.
- ☐ D) Diseñar el código teniendo en cuenta como lo podría probar.
- ☐ E) Todas las anteriores.

• Diagrama de Clases

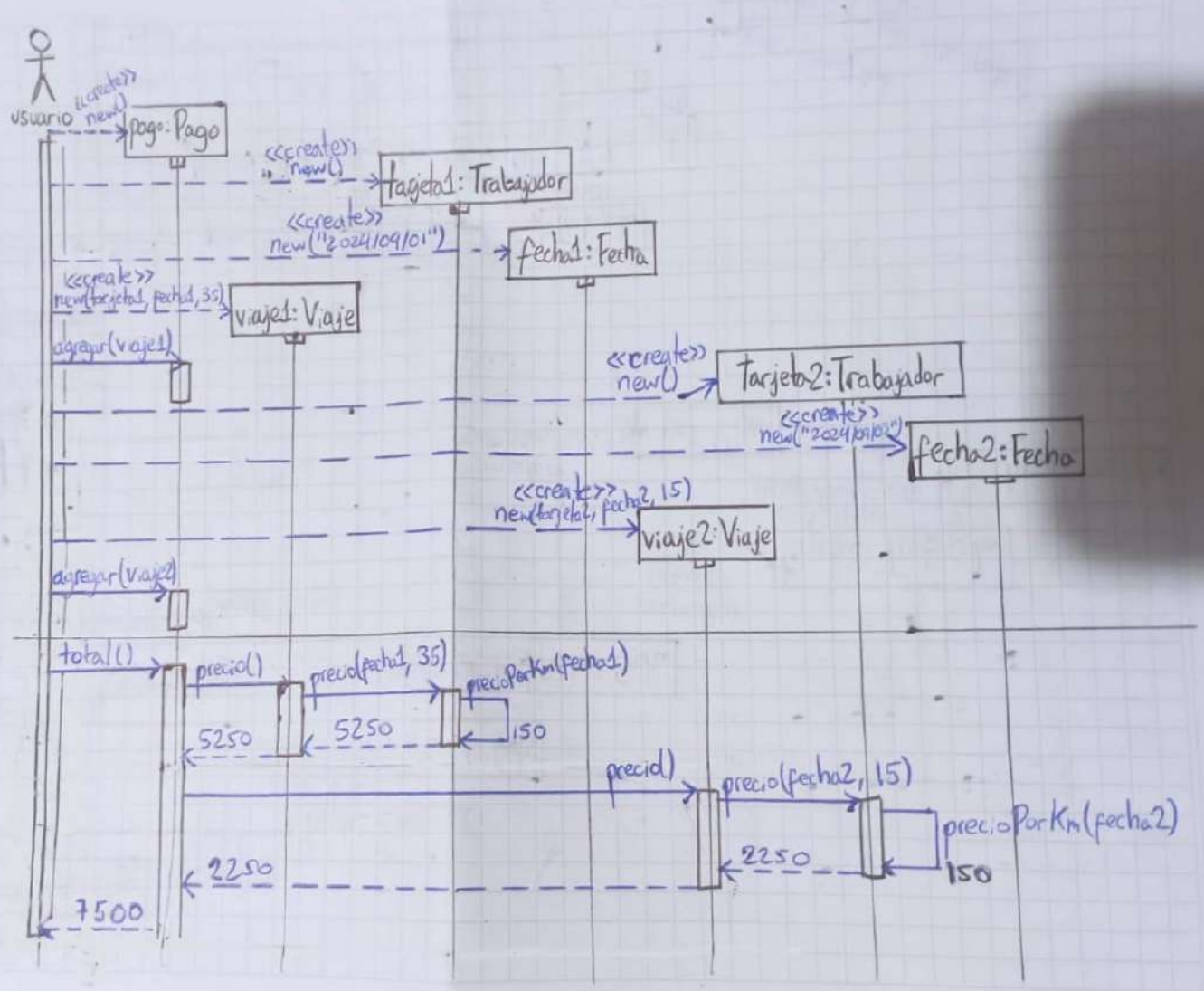


Usuario & Debe poder ver los viajes que hizo

Viaje debería delegar en tarjeta el cálculo del precio, viola principio "Tell, don't ask"

Ejercicio 1

• Diagrama de Secuencia: Escenario A



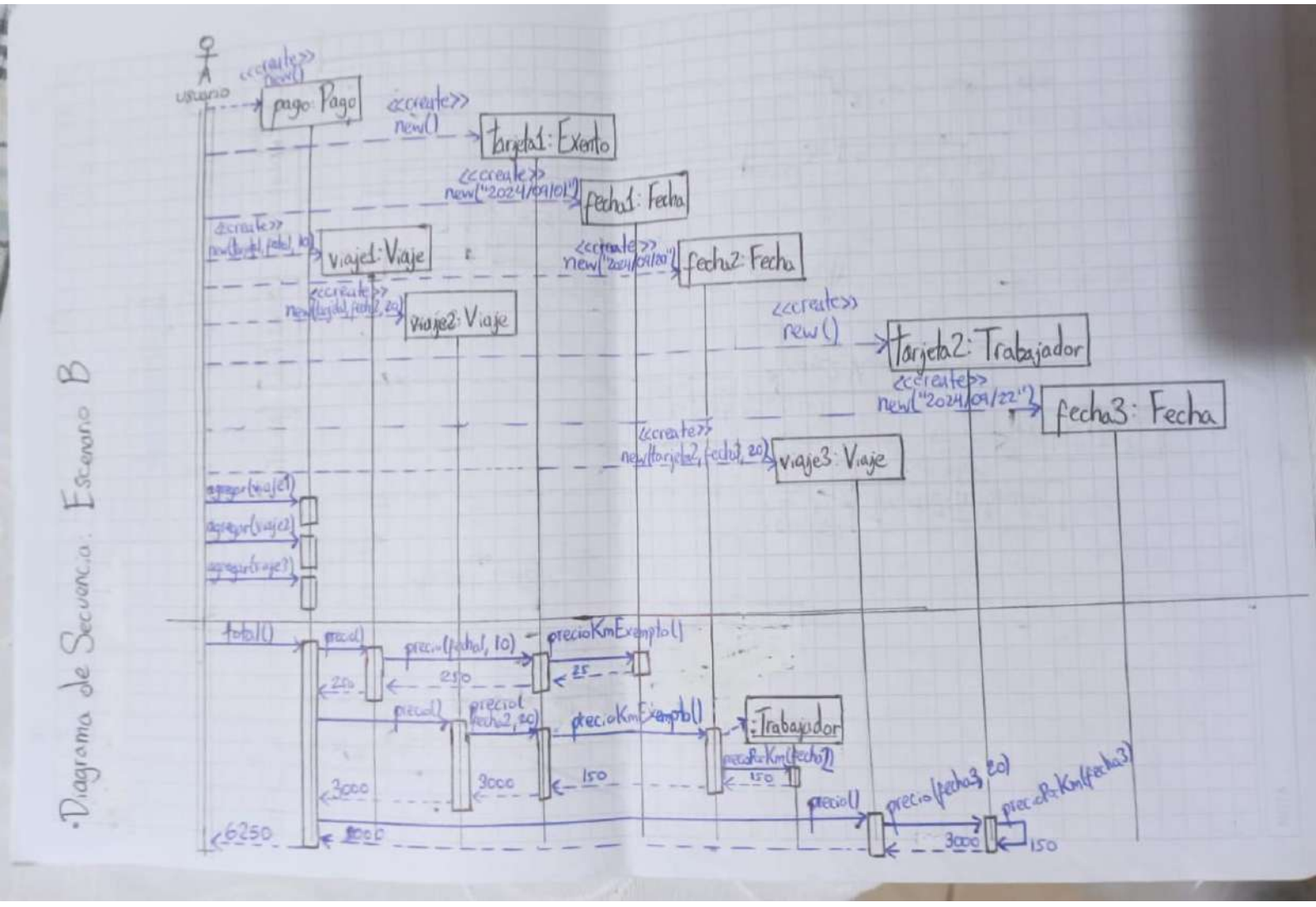


Diagrama de Secuencia: Escenario B
usuario
pago: Pago
tarjeta1: Exento
fecha1: Fecha
viaje1: Viaje
viaje2: Viaje
fecha2: Fecha
tarjeta2: Trabajador
fecha3: Fecha
viaje3: Viaje
<<create>>
new()
new("2024/09/01")
new("2024/09/20")
new("2024/09/22")
agregar(viaje1)
agregar(viaje2)
agregar(viaje3)
total()
precio()
precio(fecha1, 10)
precioKmExento()
precio(fecha2, 20)
precioKm(fecha2)
precio(fecha3, 20)
precioKm(fecha3)
: Trabajador
250
25
3000
150
6250

un programador, espero que la clase "Calculadora" sea la que se especialice en hacer cálculos, como las sumas. No tengo que asumir que un "Precio" tiene la capacidad de hacer operaciones matemáticas.

→ Encapsulamiento

(A) Elgiría el diagrama 2. El motivo principal es el principio "Tell, Don't Ask". El mismo se viola en el diagrama 1. Además, el flujo de una secuencia es obvio viendo el diagrama 2. Por ejemplo: introduzco dos precios en mi programa (p1 y p2). Luego calculo su suma con la calculadora. La misma no pretende preguntarle información (de implementación) al precio, así que simplemente le dice ("tell") que se sume p1 con p2. En ningún momento calculadora obtuvo información de cómo se organizan los precios internamente, lo cual facilita la tarea de programar.

(B) En el diagrama 1 hay altísimo acoplamiento entre las clases. Esto podría llegar a comprometer el principio de "Single Responsibility". Si cambiamos la implementación de precio, calculadora va a sufrir muchos cambios. (Por ejemplo, empezar a usar un Float). Dependiendo del contexto del programa, se podría estar
...

violando el principio de "Inversión de Dependencia". Una calculadora podría trabajar con una abstracción de alto nivel en vez de la clase precio. Por ejemplo, precio podría implementar "Número", una interfaz con el método sumar(). La calculadora puede depender de esta interfaz. Por último, se mencionó el principio "Tell, Don't Ask". En la solución 1, podemos imaginarnos que Calculadora le pregunta muchas cosas a precio (getValor(), getDecimales()) para poder hacer la suma. En todos estos ejemplos, se compromete la abstracción, ya que precio expone demasiado de sí mismo mediante excesivos getters y setters.

3 - Ejercicio de lecturas obligatorias (contestar sobre esta hoja, no es necesario justificar)
Ponderación: 15%

"Design Principles Behind Smalltalk", Dan Ingalls

- ☐ A) Un lenguaje debe diseñarse en torno a una poderosa *metáfora* que pueda aplicarse uniformemente en todas las áreas.
- ☐ B) Cuando un sistema está realmente bien construido, tanto los usuarios como los implementadores disponen de un gran *apalancamiento*.
- ☐ C) Ningún componente de un sistema *complejo* debe depender de los detalles internos de cualquier otro componente.
- ☐ D) Opciones A, B y C son correctas.
- ■ E) Opciones A y C son correctas.

Según el artículo "Unit Testing Guidelines", Petroware

- ☐ A) No define que significa unidad de prueba (unit test).
- ☐ B) La unidad de prueba tiene que ser pequeña y ejecutarse rápidamente.
- ☐ C) Las unidades de pruebas tienen que ser independientes entre sí.
- ☐ D) Diseñar el código teniendo en cuenta como lo podría probar.
- ■ E) Todas las anteriores.