Juan Menduiña

Trabajo Práctico

Ejercicio 1: *Wallpost*.

Primera parte:

Se está construyendo una red social como Facebook o Twitter. Se debe definir una clase WallPost con los siguientes atributos: un texto que se desea publicar, cantidad de likes ("me gusta") y una marca que indica si es destacado o no. La clase es subclase de Object.

Para realizar este ejercicio, utilizar el recurso que se encuentra en el sitio de la cátedra (o que puede descargar desde acá). Para importar el proyecto, seguir los pasos explicados en el documento "Trabajando con proyectos Maven".

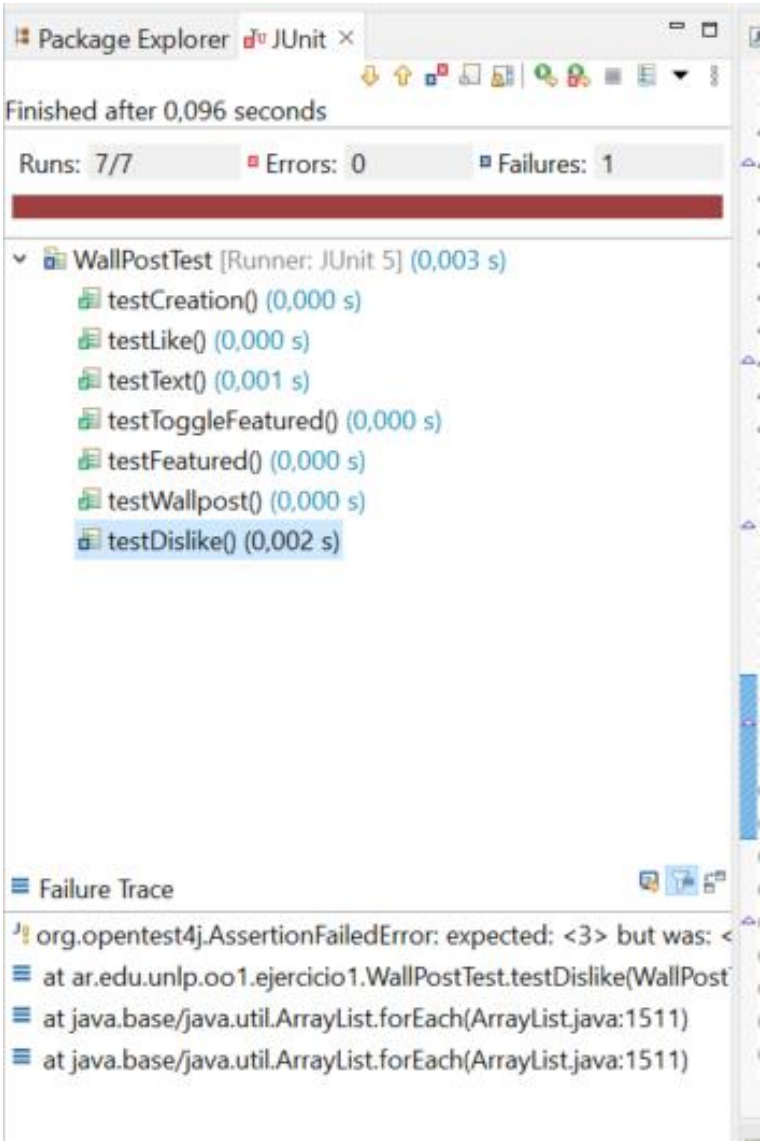
Una vez importado, dentro del mismo, se debe completar la clase WallPost de acuerdo a la siguiente especificación:

(Ver Trabajos Prácticos (P).pdf).

Segunda parte:

Utilizar los tests provistos por la cátedra para comprobar que la implementación de Wallpost es correcta. Estos se encuentran en el mismo proyecto, en la carpeta test, clase WallPostTest.

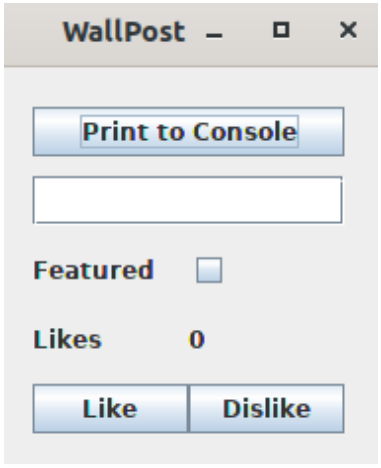
Para ejecutar los tests, simplemente, hacer click derecho sobre el proyecto y utilizar la opción Run As >> JUnit Test. Al ejecutarlo, se abrirá una ventana con el resultado de la evaluación de los tests. Sentirse libre de investigar la implementación de la clase de test. Ya se verá, en detalle, cómo implementarlas.



En el informe, Runs indica la cantidad de test que se ejecutaron. En Errors, se indica la cantidad que dieron error y, en Failures, se indica la cantidad que tuvieron alguna falla, es decir, los resultados no son los esperados. Abajo, se muestra el Failure Trace del test que falló. Si se selecciona, mostrará el mensaje de error correspondiente a ese test, que ayudará a encontrar la falla. Si se hace click sobre alguno de los test, se abrirá su implementación en el editor.

<u>Tercera parte:</u>

Una vez que la implementación pasa los tests de la primera parte, se puede utilizar la ventana que se muestra a continuación, la cual permite inspeccionar y manipular el post (definir su texto, hacer like / dislike y marcarlo como destacado).



Para visualizar la ventana, sobre el proyecto, usar la opción del menú contextual Run As >> Java Application. La ventana permite cambiar el texto del post, incrementar la cantidad de likes, etc. El botón Print to Console imprimirá los datos del post en la consola.

Ver proyecto “OO1_E1” de Java.

Ejercicio 2: Balanza Electrónica.

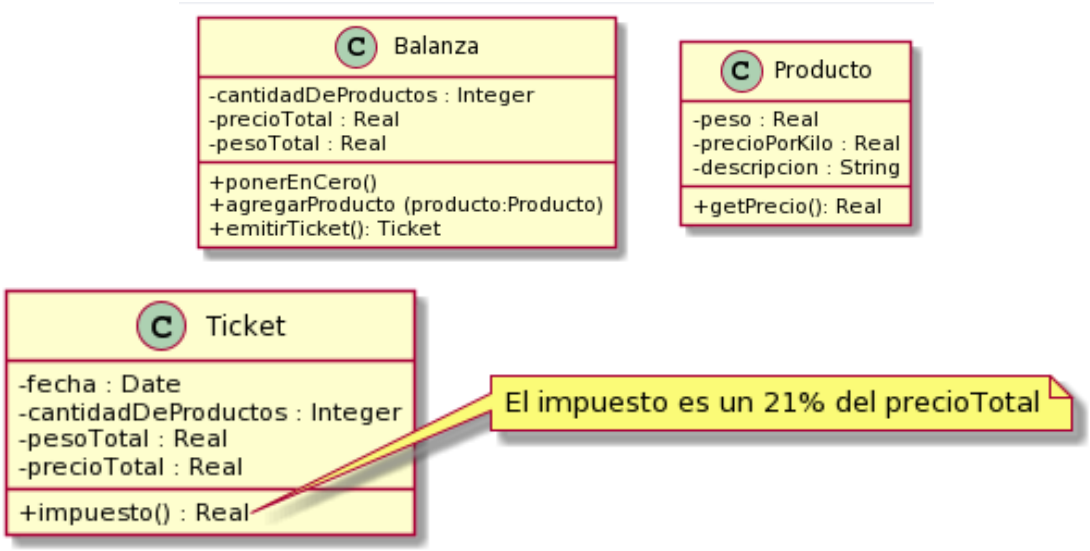
En el taller de programación, se programó una balanza electrónica. Se volverá a programar, con algún requerimiento adicional.

En términos generales, la Balanza electrónica recibe productos (uno a uno) y calcula dos totales: peso total y precio total. Además, la balanza puede poner en cero todos sus valores.

La balanza no guarda los productos. Luego, emite un ticket que indica el número de productos considerados, peso total, precio total.

(a) *Implementación:*

Crear un nuevo proyecto Maven llamado balanzaElectronica, siguiendo los pasos del documento "Trabajando con proyectos Maven, crear un proyecto Maven nuevo". En el paquete correspondiente, programar las clases que se muestran a continuación.



Observar que no se documentan en el diagrama los mensajes que nos permiten obtener y establecer los atributos de los objetos (accessors). Aunque no se incluyeron, se verá que los tests fallan si no se los implementa. Consultar con el ayudante para identificar, a partir de los tests que fallan, cuáles son los accessors necesarios (pista: todos menos los setters de balanza).

Todas las clases son subclases de Object.

Nota: Para las fechas, se utilizará la clase java.time.LocalDate. Para crear la fecha actual, se puede utilizar LocalDate.now(). También es posible crear fechas distintas a la actual. Se puede investigar más sobre esta clase en https://docs.oracle.com/en/java/javase/11/docs/api/index.html.

(b) *Probar implementación:*

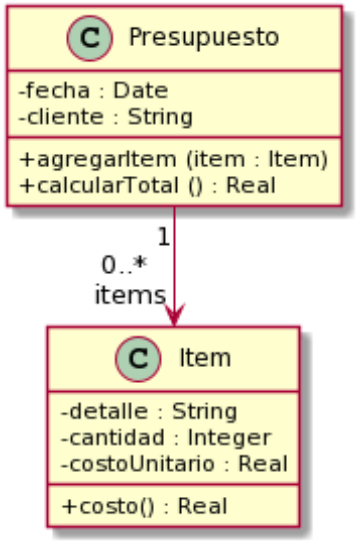
Para realizar este ejercicio, utilizar el recurso que se encuentra en el sitio de la cátedra o que puede descargarse desde este link. En este caso, se trata de dos clases, BalanzaTest y ProductoTest, las cuales se deben agregar dentro del paquete tests. Hacer las modificaciones necesarias para que el proyecto no tenga errores. Si todo salió bien, la implementación debería pasar las pruebas que definen las clases agregadas en el paso anterior. El propósito de estas clases es ejercitar una instancia de la clase Balanza y verificar que se comporta correctamente.

Ver proyecto “OO1_E2” de Java.

Ejercicio 3: Presupuestos.

Un presupuesto se utiliza para detallar los precios de un conjunto de productos que se desean adquirir. Se realiza para una fecha específica y es solicitado por un cliente, proporcionando una visión de los costos asociados.

El siguiente diagrama muestra un diseño para este dominio:



(a) *Implementación:*

Definir el proyecto Ejercicio 3 - Presupuesto y, dentro de él, implementar las clases que se observan en el diagrama. Ambas son subclases de Object.

(b) *Discutir y reflexionar:*

Prestar atención a los siguientes aspectos:

- *¿Cuáles son las variables de instancia de cada clase?*
- *¿Qué variables se inicializan? ¿De qué formas se puede realizar esta inicialización?*
- *¿Qué ventajas y desventajas se encuentran en cada una de ellas?*

(c) *Probar implementación:*

Utilizar los tests provistos para confirmar que la implementación ofrece la funcionalidad esperada. En este caso, se trata de dos clases: ItemTest y PresupuestoTest, que se deben agregar dentro del paquete tests. Hacer las modificaciones necesarias para que el proyecto no tenga errores. Sentirse libre de explorar las clases de test para intentar entender qué es lo que hacen.

Ver proyecto “OO1_E3” de Java.

Ejercicio 4: Balanza Mejorada.

Realizando el ejercicio de los presupuestos, se aprendió que un objeto puede tener una colección de otros objetos. Con esto en mente, ahora, se quiere mejorar la balanza implementada en el Ejercicio 2.

(a) *Mejorar la balanza para que recuerde los productos ingresados (los mantenga en una colección). Analizar de qué forma puede realizarse este nuevo requerimiento e implementar el mensaje public List<Producto> getProductos(), que retorna todos los productos ingresados a la balanza (en la compra actual, es decir, desde la última vez que se la puso a cero).*

- *¿Qué cambio produce este nuevo requerimiento en la implementación del mensaje ponerEnCero() ?*
- *¿Es necesario, ahora, almacenar los totales en la balanza? ¿Se pueden obtener estos valores de otra forma?*

(b) *Con esta nueva funcionalidad, se puede enriquecer al Ticket, haciendo que él también conozca a los productos (a futuro, se podría imprimir el detalle). Ticket también debería entender el mensaje public List<Producto> getProductos().*

- *¿Qué cambios se creen necesarios, en Ticket, para que pueda conocer a los productos?*
- *¿Estos cambios modifican las responsabilidades ya asignadas de realizar el cálculo del precio total? ¿El ticket adquiere nuevas responsabilidades que antes no tenía?*

(c) *Después de hacer estos cambios, ¿siguen pasando los tests? ¿Está bien que sea así?*

Ver proyecto “OO1_E4” de Java.

<u>Ejercicio 5:</u> Inversores. (*)

Estamos desarrollando una aplicación móvil para que un inversor pueda conocer el estado de sus inversiones. El sistema permite manejar dos tipos de inversiones: inversión en acciones e inversión en plazo fijo. En todo momento, se desea poder conocer el valor actual de cada inversión y de las inversiones realizadas por el inversor.

Para las inversiones en acciones, el valor actual se calcula multiplicando el valor unitario de una acción por la cantidad de acciones que se posee. De las acciones, se conoce el nombre que las identifica en el mercado de valores y que un inversor puede invertir en diferentes acciones con diferentes valores unitarios. Por su parte, para los plazo fijos, el valor actual consiste en el cálculo del valor inicial de constitución del plazo fijos sumando los intereses diarios desde la fecha de constitución hasta hoy.

De las inversiones en acciones, es importante poder conocer su nombre, la cantidad de acciones en las que se invertirá y el valor unitario de cada acción. Por su parte, los plazos fijos se constituyen en una fecha, es importante conocer el monto depositado y cuál es el porcentaje de interés que genera.

Por último, el valor de inversión actual de un inversor es la suma de los valores actuales de todas las inversiones que posee. Un inversor puede agregar y sacar inversiones de su cartera de inversiones cuando lo desee. Las inversiones pueden ser tanto en acciones como en plazo fijos y pueden estar mezcladas.

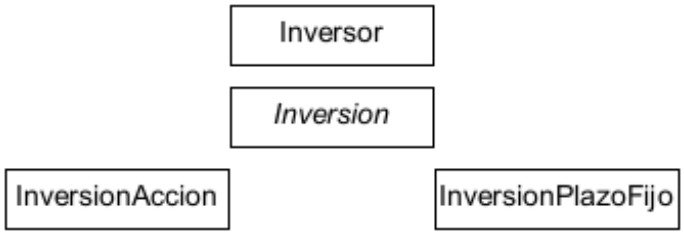
(a) *Realizar la lista de conceptos candidatos. Clasificar cada concepto dentro de las categorías vistas en la teoría.*

Lista de conceptos candidatos:

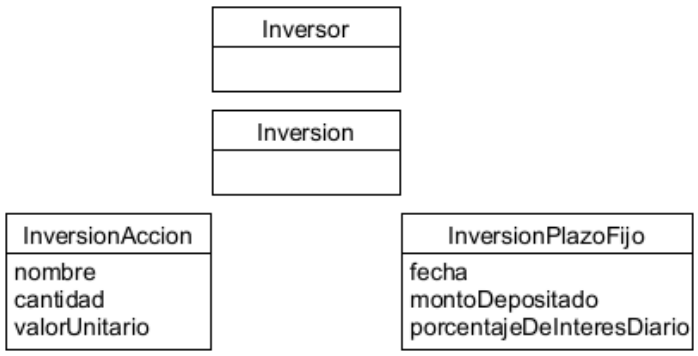
1. Objeto físico o tangible: Objeto físico o tangible relevante.
2. Especificación de una cosa: Características esenciales de objetos.
3. Lugar: Lugar relevante.
4. Transacción: Acontecimientos donde algo cambia en el sistema.
5. Roles de la gente: Personas o roles dentro del sistema.
6. Contenedor de cosas: Agrupa o contiene objetos.
7. Cosas de un contenedor: Son los objetos dentro del contenedor anterior.
8. Otros sistemas: Con los que se podría conectar, si existieran.
9. Hechos: Eventos o datos que pueden cambiar.
10. Reglas y políticas: Relacionadas con la lógica del negocio.
11. Registros financieros/laborales: Datos relevantes del ámbito económico.
12. Manuales/documentos: Manuales/documentos relevantes.

Categoría de clase conceptual	Clases conceptuales
Objeto físico o tangible	aplicación móvil inversor
Especificación de una cosa	nombre de la acción cantidad de acciones valor unitario de la acción fecha de constitución del plazo fijo monto depositado en el plazo fijo porcentaje de interés que genera el plazo fijo
Lugar	-
Transacción	registro de una inversión (compra de acción, constitución de plazo fijo) cálculo del valor de las acciones cálculo del valor del plazo fijo cálculo del valor actual de la cartera
Roles de la gente	inversor
Contenedor de cosas	inversor
Cosas de un contenedor	inversión inversión en acciones inversión en plazo fijo
Otros sistemas	-
Hechos	intereses ganados valor actual de la inversión suma del valor actual de todas las inversiones del inversor
Reglas y políticas	regla de cálculo del valor actual de las acciones regla de cálculo del valor actual del plazo fijo regla de cálculo del valor actual de la cartera
Registros financieros/laborales	-
Manuales, documentos	-

(b) *Graficar el modelo de dominio usando UML.*



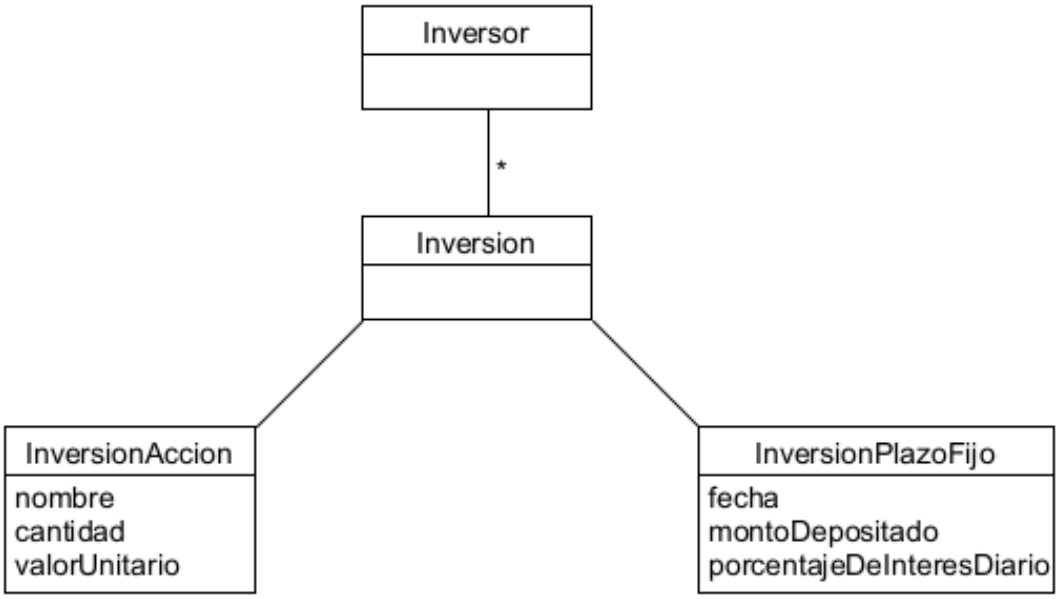
(c) *Actualizar el modelo de dominio incorporando los atributos a los conceptos.*



(d) *Agregar asociaciones entre conceptos, indicando, para cada una de ellas, la categoría a la que pertenece, de acuerdo a lo explicado en la teoría, y demás atributos, según sea necesario.*

Lista de asociaciones:

Categoría	Asociaciones
A es una parte física de B	
A es una parte lógica de B	
A está físicamente contenido en B	
A está lógicamente contenido en B	Inversión Acción - Inversión Inversión Plazo Fijo - Inversión
A es una descripción para B	
A es un miembro de B	
A usa o maneja a B	
A se comunica con B	
A está relacionado con la transacción B	
A es una transacción relacionada con otra transacción B	
A es dueño de B	Inversor - Inversión


Inversor
*
Inversion
InversionAccion
nombre
cantidad
valorUnitario
InversionPlazoFijo
fecha
montoDepositado
porcentajeDeInteresDiario

Ejercicio 6: Distribuidora Eléctrica. (*)

Una distribuidora eléctrica desea gestionar los consumos de sus usuarios para la emisión de facturas de cobro.

De cada usuario, se conoce su nombre y domicilio. Se considera que cada usuario sólo puede tener un único domicilio en donde se registran los consumos.

Los consumos de los usuarios se dividen en dos componentes:

- *Consumo de energía activa: Tiene un costo asociado para el usuario. Se mide en kWh (kilowatt/hora).*
- *Consumo de energía reactiva: No genera ningún costo para el usuario, es decir, se utiliza sólo para determinar si hay alguna bonificación. Se mide en kVArh (kilo voltio-amperio reactivo hora).*

Se cuenta con un cuadro tarifario que establece el precio del kWh para calcular el costo del consumo de energía activa. Este cuadro tarifario puede ser ajustado, periódicamente, según sea necesario (por ejemplo, para reflejar cambios en los costos).

Para emitir la factura de un cliente, se tiene en cuenta sólo su último consumo registrado. Los datos que debe contener la factura son los siguientes:

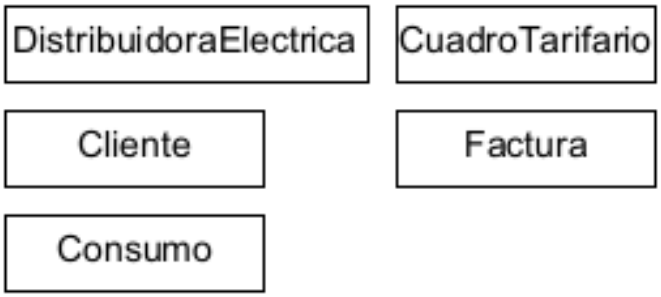
- *El usuario a quien se está cobrando.*
- *La fecha de emisión.*
- *La bonificación, si aplica.*
- *El monto final de la factura: Se calcula restando la bonificación al costo del consumo:*
  - *El costo del consumo se calcula multiplicando el consumo de energía activa por el precio del kWh proporcionado por el cuadro tarifario.*
  - *Se calcula su factor de potencia para determinar si hay alguna bonificación aplicable. Si el factor de potencia estimado (fpe) del último consumo del usuario es mayor a 0,8, el usuario recibe una bonificación del 10%.*

(a) *Realizar la lista de conceptos candidatos.*

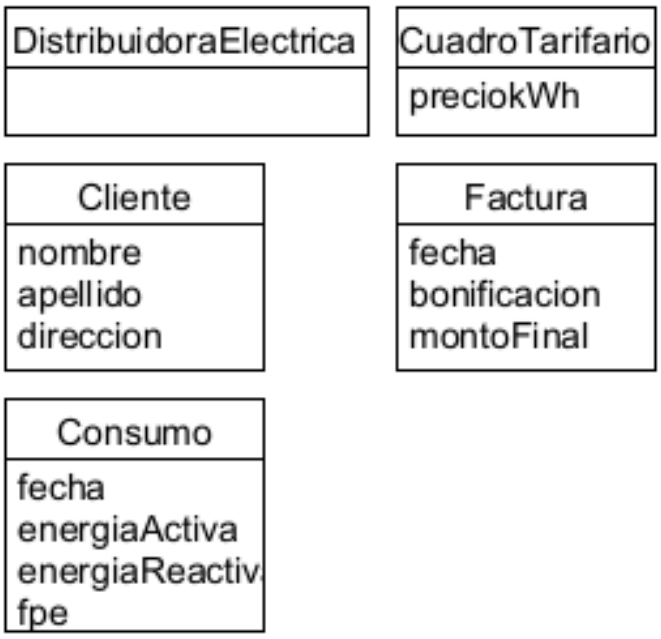
Lista de conceptos candidatos:

Categoría de clase conceptual	Clases conceptuales
Objeto físico o tangible	distribuidora eléctrica cliente factura
Especificación de una cosa	nombre, apellido y dirección del cliente fecha del consumo energía activa del consumo energía reactiva del consumo factor de potencia estimado (fpe) precio del kWh actual cliente de la factura último consumo registrado precio del kWh de la factura bonificación de la factura monto final de la factura
Lugar	distribuidora eléctrica
Transacción	actualización del cuadro tarifario emisión de una factura registro de un consumo cálculo del factor de potencia estimado cálculo del costo del consumo determinación de bonificación cálculo del monto final de la factura
Roles de la gente	cliente
Contenedor de cosas	distribuidora eléctrica cliente factura
Cosas de un contenedor	cuadro tarifario de la distribuidora eléctrica cliente de la distribuidora eléctrica consumo del cliente cliente de la factura consumo de la factura
Otros sistemas	-
Hechos	precio del kWh actual cliente de la factura último consumo registrado (fecha, energía activa, energía reactiva, fpe) precio del kWh de la factura bonificación de la factura monto final de la factura
Reglas y políticas	regla de que la factura se emite sólo con el último consumo registrado regla del cálculo del costo del consumo política de bonificación (fpe $>$ 0,8)
Registros financieros/laborales	factura
Manuales, documentos	cuadro tarifario

(b) *Graficar el modelo de dominio usando UML.*



(c) *Actualizar el modelo de dominio incorporando los atributos a los conceptos.*



(d) *Agregar asociaciones entre conceptos, indicando, para cada una de ellas, la categoría a la que pertenece, de acuerdo a lo explicado en la teoría, y demás atributos, según sea necesario.*

Lista de asociaciones:

Categoría	Asociaciones
A es una parte física de B	
A es una parte lógica de B	
A está físicamente contenido en B	
A está lógicamente contenido en B	Cuadro Tarifario - Distribuidora Eléctrica Cliente - Distribuidora Eléctrica Consumo - Cliente Cliente - Factura Consumo - Factura
A es una descripción para B	
A es un miembro de B	
A usa o maneja a B	Factura - Cuadro Tarifario
A se comunica con B	
A está relacionado con la transacción B	
A es una transacción relacionada con otra transacción B	
A es dueño de B	

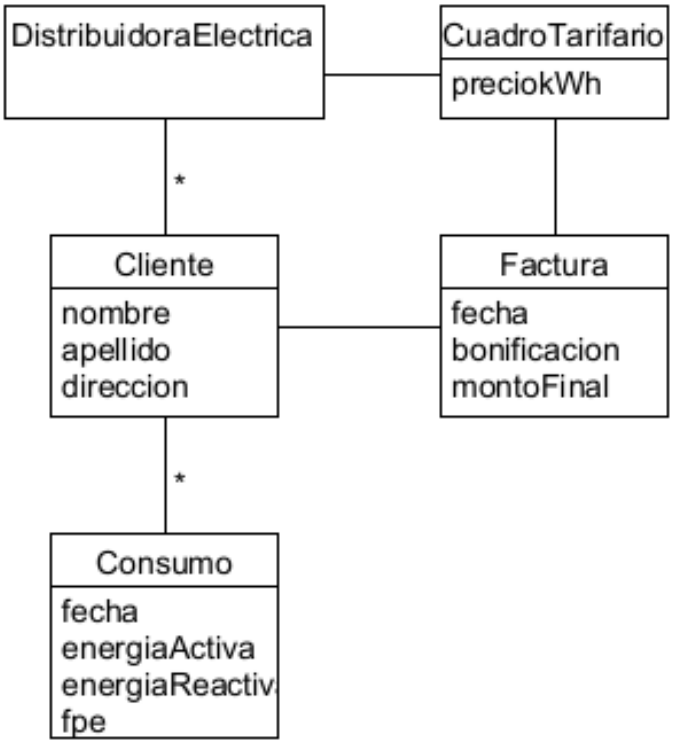
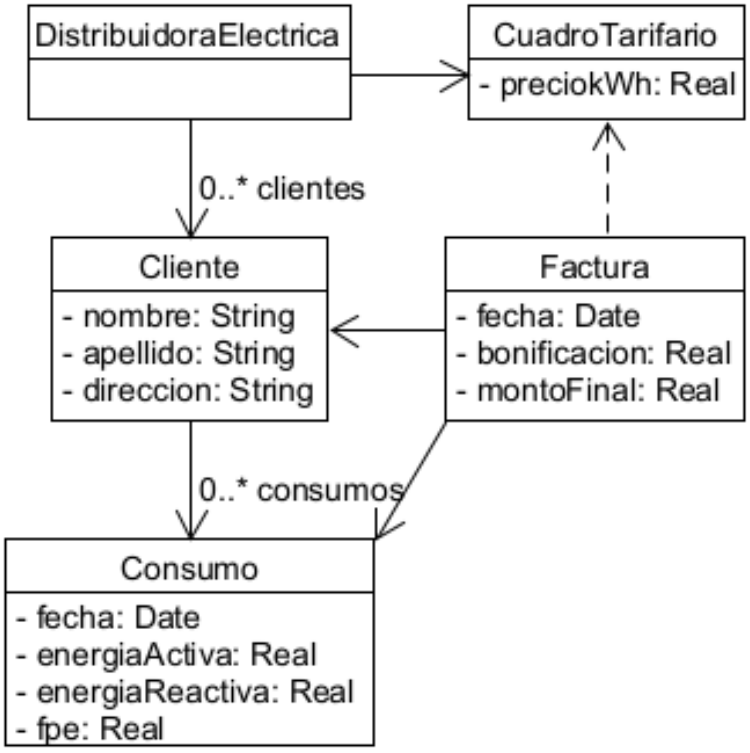


Diagrama de clases UML (adicional):

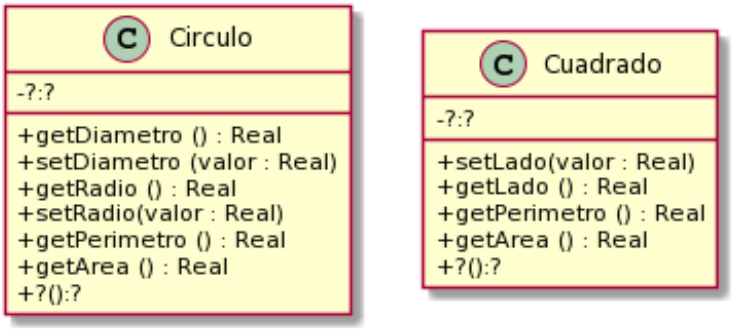


Ejercicio 7: Figuras y Cuerpos.

Figuras en 2D:

En Taller de Programación, se definieron clases para representar figuras geométricas. Se retomará ese ejercicio para trabajar con Cuadrados y Círculos.

El siguiente diagrama de clases documenta los mensajes que estos objetos deben entender.



Fórmulas y mensajes útiles:

- *Diámetro del círculo: radio * 2.*
- *Perímetro del círculo: π * diámetro.*
- *Área del círculo: π * $radio^2$.*
- *π se obtiene enviando el mensaje #pi a la clase Float (Float pi) (ahora Math.PI).*

(a) *Implementación:*

Definir un nuevo proyecto figurasYCuerpos. Implementar las clases Círculo y Cuadrado, siendo ambas subclases de Object. Decidir qué variables de instancia son necesarias. Se pueden agregar mensajes adicionales si se cree necesario.

(b) *Discutir y reflexionar:*

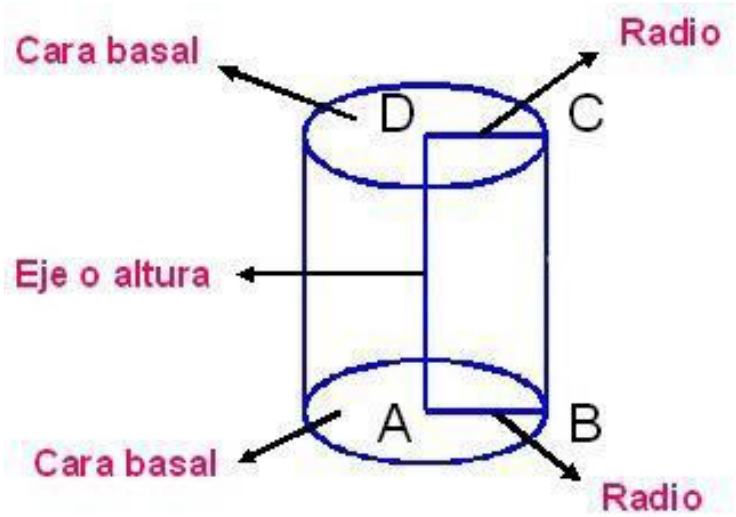
¿Qué variables de instancia se definieron? ¿Se puede hacer de otra manera? ¿Qué ventajas se encuentran en la forma en que se realizó?

Ver proyecto "OO1_E7" de Java.

Cuerpos en 3D:

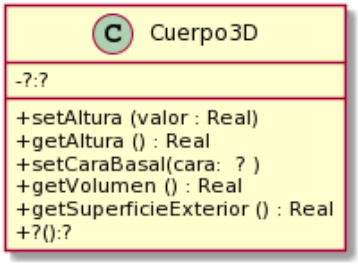
Ahora que se tiene Círculos y Cuadrados, se pueden usar para construir cuerpos (en 3D) y calcular su volumen y superficie o área exterior.

- *Se va a pensar a un cilindro como "un cuerpo que tiene una figura 2D como cara basal y que tiene una altura (ver la siguiente imagen)". Es decir, si, en el lugar de la figura2D, se tuviera un círculo, se formaría el siguiente cuerpo 3D.*



- *Si se reemplaza la cara basal por un rectángulo, se tendrá un prisma (una caja de zapatos).*

El siguiente diagrama de clases documenta los mensajes que entiende un cuerpo3D:



Fórmulas útiles:

- *El área o superficie exterior de un cuerpo es: 2 * área-cara-basal + perímetro-cara-basal * altura-del-cuerpo.*
- *El volumen de un cuerpo es: área-cara-basal * altura.*

Más info interesante: A la figura que da forma al cuerpo (el círculo o el cuadrado en nuestro caso) se le llama directriz. Y a la recta en la que se mueve se le llama generatriz. En Wikipedia (Cilindro), se puede aprender un poco más al respecto.

(a) *Implementación:*

Implementar la clase Cuerpo 3D, la cual es subclase de Object. Decidir qué variables de instancia son necesarias. También decidir si es necesario hacer cambios en las figuras 2D.

(b) *Probar implementación:*

Siguiendo los ejemplos de ejercicios anteriores, ejecutar las pruebas automatizadas provistas. En este caso, se trata de tres clases (CuerpoTest, TestCirculo y TestCuadrado) que se deben agregar dentro del paquete tests. Hacer las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consultar al ayudante.

(c) *Discutir y reflexionar:*

Discutir con el ayudante las elecciones de variables de instancia y métodos adicionales. ¿Es necesario todo lo que se definió?

Ver proyecto “OO1_E7” de Java.

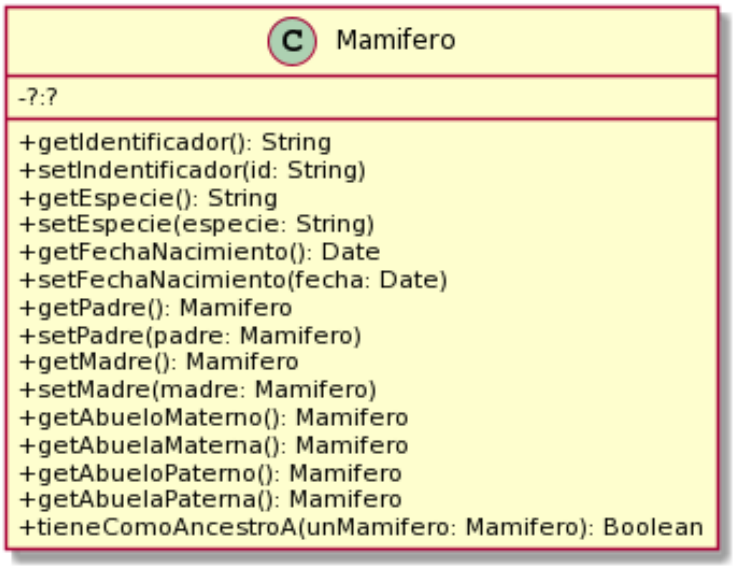
Ejercicio 8: Genealogía Salvaje.

En una reserva de vida salvaje (como la estación de cría ECAS, en el camino Centenario), los cuidadores quieren llevar registro detallado de los animales que cuidan y sus familias. Para ello, han pedido ayuda. Se debe:

(a) *Completar el diseño e implementar:*

Modelar una solución en objetos e implementar la clase Mamífero (como subclase de Object). El siguiente diagrama de clases (incompleto) nos da una idea de los mensajes que un mamífero entiende.

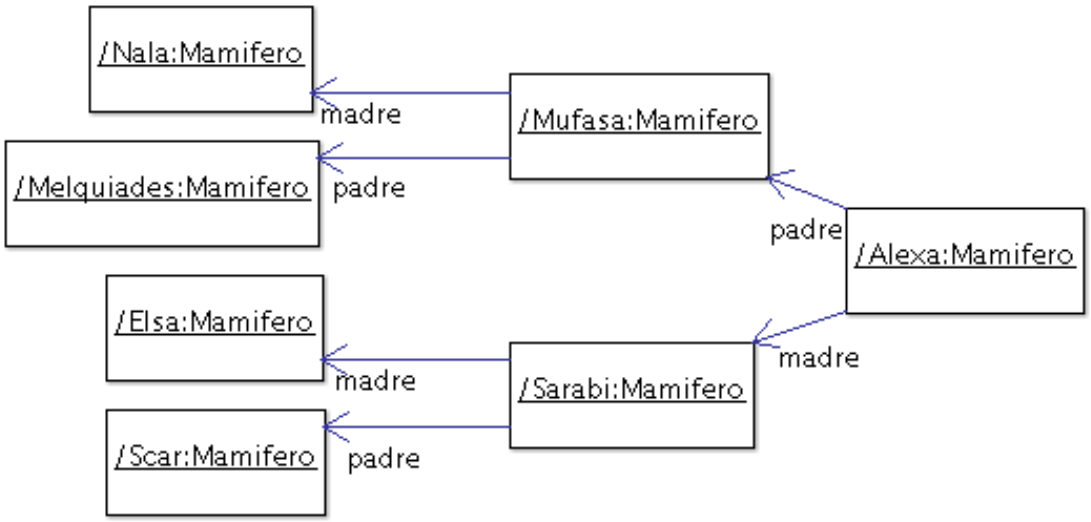
Proponer una solución para el método tieneComoAncestroA(...) y dejar la implementación para el final y discutir la solución con el ayudante.



Completar el diagrama de clases para reflejar los atributos y relaciones requeridas en la solución.

(b) *Pruebas automatizadas:*

Siguiendo los ejemplos de ejercicios anteriores, ejecutar las pruebas automatizadas provistas. En este caso, se trata de una clase, MamiferoTest, que se debe agregar dentro del paquete tests. En esta clase, se trabaja con la familia mostrada en la siguiente figura:



En el diagrama, se puede apreciar el nombre/identificador de cada uno de ellos (por ejemplo, Nala, Mufasa, Alexa, etc). Hacer las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consultar al ayudante.

Ver proyecto “OO1_E8” de Java.

Ejercicio 9: Red de Alumbrado. (*)

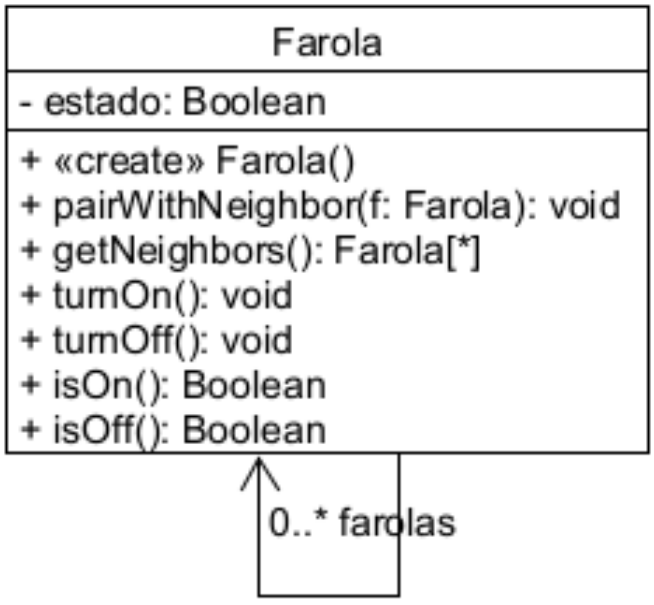
Imaginar una red de alumbrado donde cada farola está conectada a una o varias vecinas formando un grafo conexo. Cada una de las farolas tiene un interruptor. Es suficiente con encender o apagar una farola cualquiera para que se enciendan o apaguen todas las demás. Sin embargo, si se intenta apagar una farola apagada (o si se intenta encender una farola encendida), no habrá ningún efecto, ya que no se propagará esta acción hacia las vecinas.

La funcionalidad a proveer permite:

1. Crear farolas (inicialmente, están apagadas).
2. Conectar farolas a tantas vecinas como uno quiera (las conexiones son bidireccionales).
3. Encender una farola (y obtener el efecto antes descrito).
4. Apagar una farola (y obtener el efecto antes descrito).

(a) *Modelar e implementar:*

(i) *Realizar el diagrama UML de clases de la solución al problema.*



(ii) *Implementar, en Java, la clase Farola, como subclase de Object, con los siguientes métodos:*

(Ver Trabajos Prácticos (P).pdf).

Ver proyecto “OO1_E9” de Java.

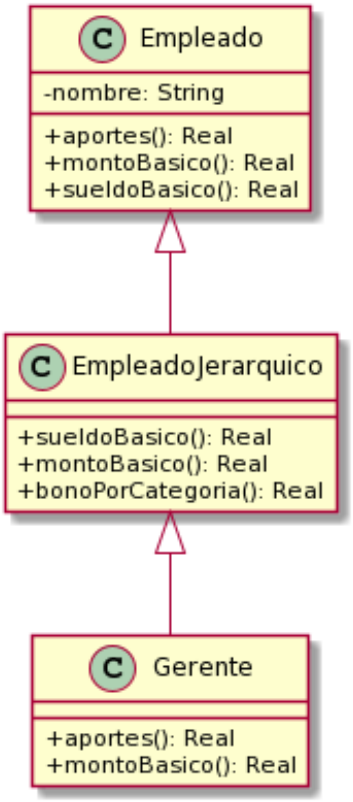
(b) *Pruebas automatizadas:*

Utilizar los tests provistos por la cátedra para probar las implementaciones del inciso (a).

Ver proyecto “OO1_E9” de Java.

Ejercicio 10: *Method Lookup* con Empleados.

Sea la jerarquía de Empleado como muestra la figura de la izquierda, cuya implementación de referencia se incluye en la tabla.



Empleado	EmpleadoJerarquico	Gerente

`public double montoBasico() {`
`return 35000;`
`}`
	`public double sueldoBasico() {`
	`return super.sueldoBasico()+ this.bonoPorCategoria();`
	`}`
		`public double aportes() {`
		`return this.montoBasico() * 0.05d;`
		`}`

`public double aportes(){`
`return 13500;`
`}`
	`public double montoBasico() {`
	`return 45000;`
	`}`
		`public double montoBasico() {`
		`return 57000;`
		`}`

`public double sueldoBasico() {`
`return this.montoBasico() +`
`this.aportes();}`
	`public double bonoPorCategoria() {`
	`return 8000;`
	`}`	

Analizar cada uno de los siguientes fragmentos de código y resolver las tareas indicadas abajo:

```
Gerente alan = new Gerente("Alan Turing");
double aportesDeAlan = alan.aportes();
```

```
Gerente alan = new Gerente("Alan Turing");
double sueldoBasicoDeAlan = alan.sueldoBasico();
```

(a) *Listar todos los métodos, indicando nombre y clase, que son ejecutados como resultado del envío del último mensaje de cada fragmento de código (por ejemplo, (1) aportes de la clase Empleado, (2) ...).*

Primer fragmento de código:

(1) *aportes* de la clase Gerente.
(2) *montoBasico* de la clase Gerente.

Segundo fragmento de código:

(1) *sueldoBasico* de la clase EmpleadoJerarquico.
(2) *sueldoBasico* de la clase Empleado.
(3) *montoBasico* de la clase Gerente.
(4) *aportes* de la clase Gerente .
(5) *bonoPorCategoria* de la clase EmpleadoJerarquico.

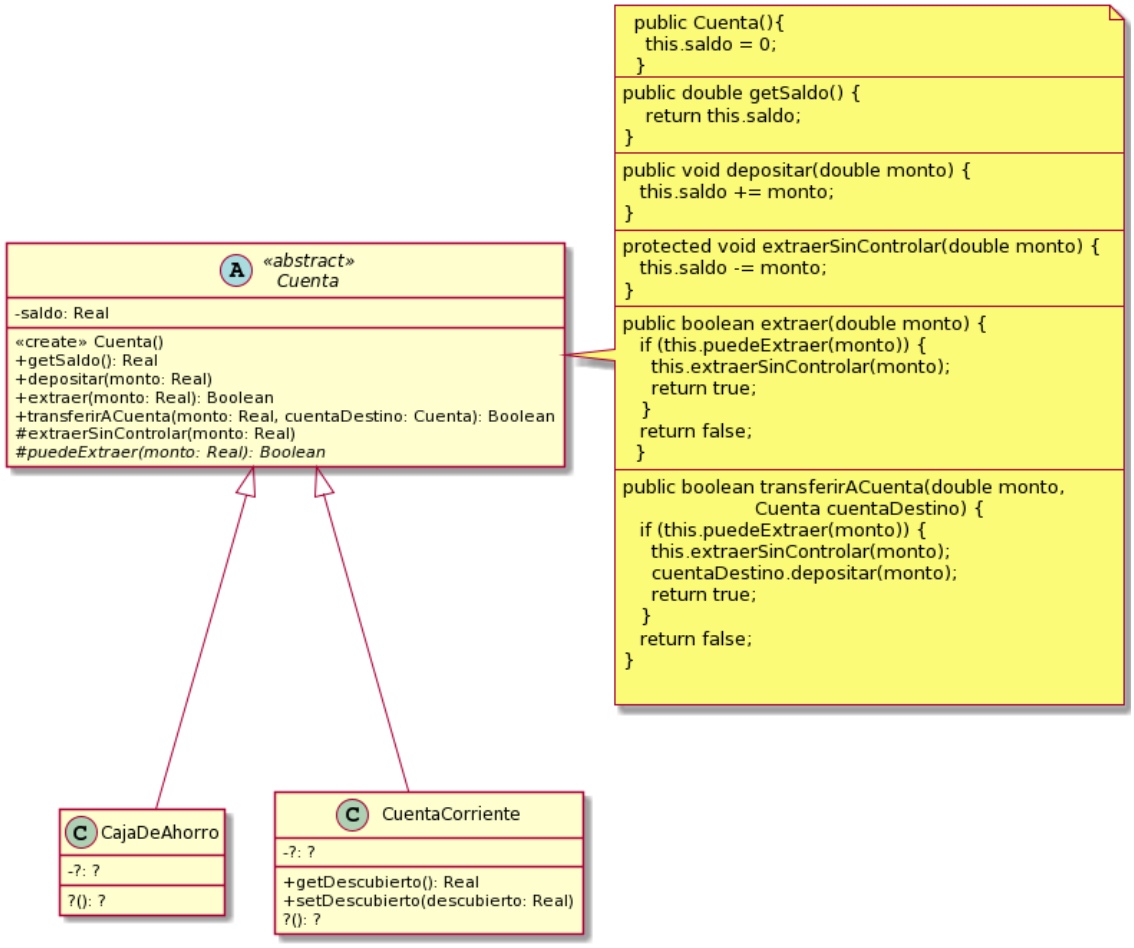
(b) *¿Qué valores tendrán las variables aportesDeAlan y sueldoBasicoDeAlan luego de ejecutar cada fragmento de código?*

Los valores que tendrán las variables *aportesDeAlan* y *sueldoBasicoDeAlan* luego de ejecutar cada fragmento de código serán 2.850 y 67.850, respectivamente.

Ver proyecto “OO1_E10” de Java.

Ejercicio 11: Cuenta con Ganchos.

Observar, con detenimiento, el diseño que se muestra en el siguiente diagrama. La clase cuenta es abstracta. El método puedeExtraer() es abstracto. Las clases CajaDeAhorro y CuentaCorriente son concretas y están incompletas.



(a) *Completar la implementación de las clases CajaDeAhorro y CuentaCorriente para que se puedan efectuar depósitos, extracciones y transferencias, teniendo en cuenta los siguientes criterios:*

- *Las cajas de ahorro sólo pueden extraer y transferir cuando cuentan con fondos suficientes.*
- *Las extracciones, los depósitos y las transferencias desde cajas de ahorro tienen un costo adicional de 2% del monto en cuestión (tenerlo en cuenta antes de permitir una extracción o transferencia desde caja de ahorro).*
- *Las cuentas corrientes pueden extraer aun cuando el saldo de la cuenta sea insuficiente. Sin embargo, no deben superar cierto límite por debajo del saldo. Dicho límite se conoce como límite de descubierto (algo así como el máximo saldo negativo permitido). Ese límite es diferente para cada cuenta (lo negocia el cliente con la gente del banco).*
- *Cuando se abre una cuenta corriente, su límite descubierto es 0 (no olvidar definir el constructor por default).*

(b) *Reflexionar, charlar con el ayudante y responder a las siguientes preguntas:*

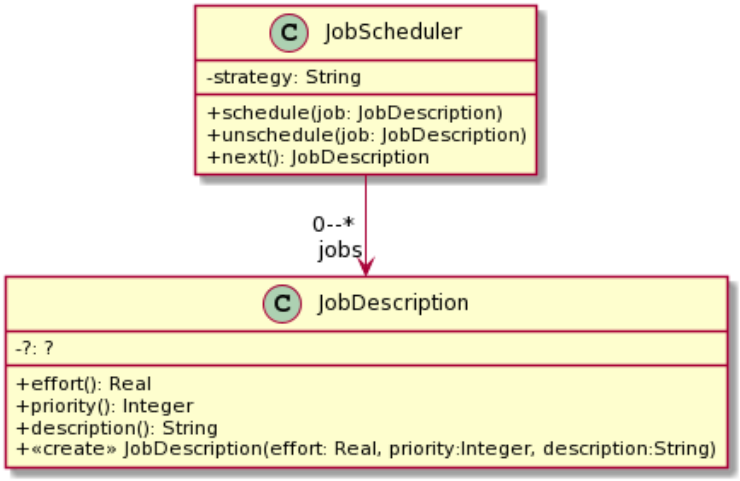
- *¿Por qué este ejercicio se llama "Cuenta con ganchos"?*
- *En las implementaciones de los métodos extraer() y transferirACuenta() que se ven en el diagrama, ¿quién es this? ¿Se puede decir de qué clase es this?*
- *¿Por qué se decide que los métodos puedeExtraer() y extraerSinControlar tengan visibilidad "protegido"?*
- *¿Se puede transferir de una caja de ahorro a una cuenta corriente y viceversa? ¿por qué? ¡Probarlo!*
- *¿Cómo se declara, en Java, un método abstracto? ¿Es obligatorio implementarlo?*
- *¿Qué dice el compilador de Java si una subclase no implementa un método abstracto que hereda?*

(c) *Escribir los tests de unidad que se crean necesarios para validar que la implementación funciona adecuadamente.*

Ver proyecto "OO1_E11" de Java.

Ejercicio 12: *Job Scheduler.*

El JobScheduler es un objeto cuya responsabilidad es determinar qué trabajo debe resolverse a continuación. El siguiente diseño ayuda a entender cómo funciona la implementación actual del JobScheduler.



- *El mensaje schedule(job: JobDescription) recibe un job (trabajo) y lo agrega al final de la colección de trabajos pendientes.*
- *El mensaje next() determina cuál es el siguiente trabajo de la colección que debe ser atendido, lo retorna y lo quita de la colección.*

En la implementación actual del método next(), el JobScheduler utiliza el valor de la variable strategy para determinar cómo elegir el siguiente trabajo.

Dicha implementación presenta dos serios problemas de diseño:

- *Secuencia de ifs (o sentencia switch/case) para implementar alternativas de un mismo comportamiento.*
- *Código duplicado.*

(a) *Analizar el código existente:*

Utilizar el código y los tests provistos por la cátedra y aplicar lo aprendido (en particular, en relación a herencia y polimorfismo) para eliminar los problemas mencionados. Sentirse libre de agregar nuevas clases como se considere necesario. También se puede cambiar la forma en la que los objetos se crean e inicializan. Asumir que una vez elegida una estrategia para un scheduler no puede cambiarse.

(b) *Pruebas automatizadas:*

Los cambios, probablemente, hagan que los tests dejen de funcionar. Corregirlos y mejorarlos como sea necesario.

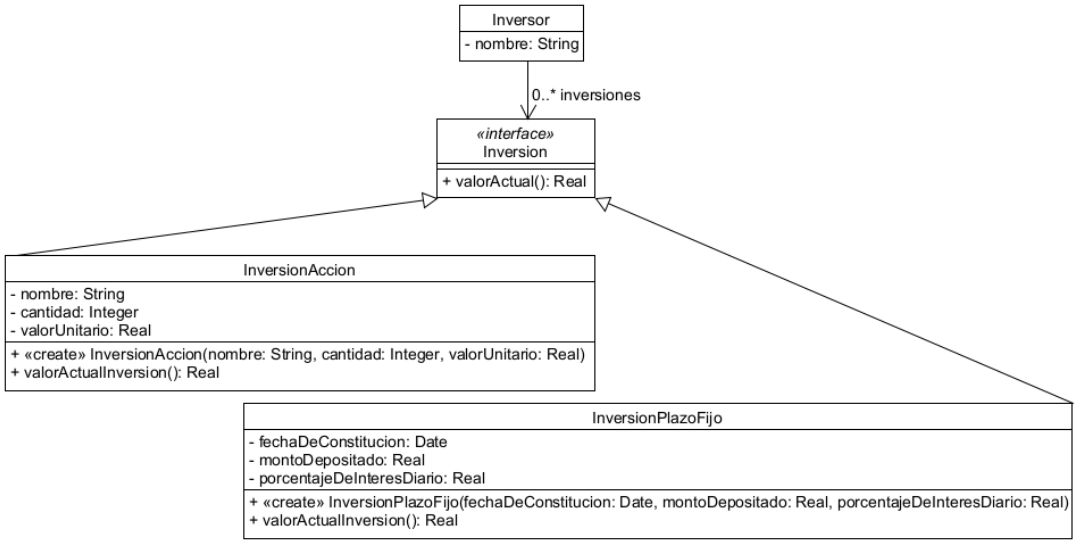
Ver proyecto “OO1_E12” de Java.

Ejercicio 13: ¡A Implementar Inversores!

Retomando el Ejercicio 5, donde se trabajó en el diseño y modelado UML de un sistema de inversiones, donde se definieron las clases, atributos y asociaciones necesarias para representar inversores y sus diferentes tipos de inversiones. Ahora, es el momento de llevar el diseño a la práctica: se va a implementar, en Java, lo diseñado y asegurar su calidad mediante pruebas automatizadas.

(a) *Implementar:*

(i) *Realizar el mapeo del modelo conceptual a un diagrama de clases de UML.*



(ii) *Implementar, en Java, lo necesario para que se pueda conocer el valor actual de cada inversión. Y también el monto total de las inversiones realizadas por un inversor.*

Ver proyecto “OO1_E13” de Java.

(b) *Pruebas automatizadas:*

> **(i)** *Diseñar los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.*
> **(ii)** *Implementar, utilizando JUnit, los tests automatizados diseñados en el inciso anterior.*

Ver proyecto “OO1_E13” de Java.

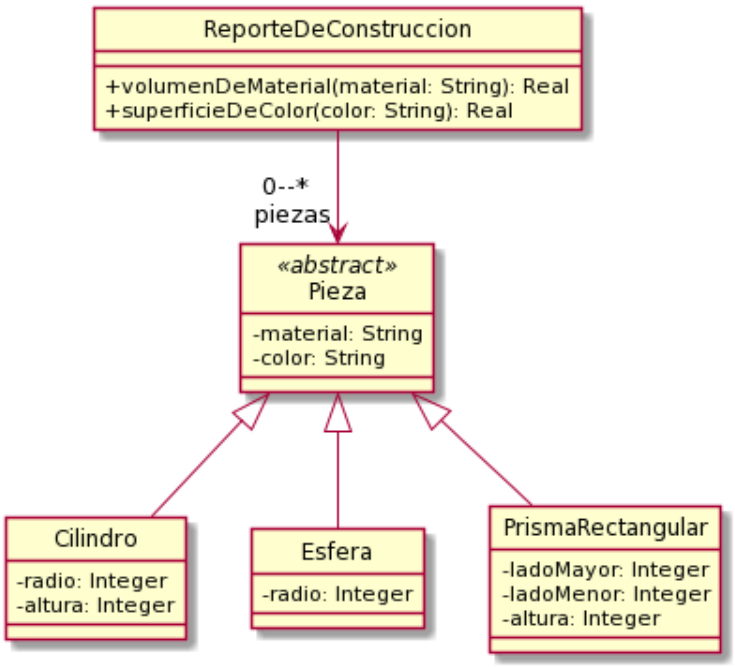
(c) *Discutir con el ayudante:*

Consultar con un ayudante los casos de prueba diseñados.

<u>Ejercicio 14:</u> Volumen y Superficie de Sólidos.

Una empresa siderúrgica quiere introducir, en su sistema de gestión, nuevos cálculos de volumen y superficie exterior para las piezas que produce. El volumen le sirve para determinar cuánto material ha utilizado. La superficie exterior le sirve para determinar la cantidad de pintura que utilizó para pintar las piezas.

El siguiente diagrama UML muestra el diseño actual del sistema. En el mismo, puede observarse que un ReporteDeConstruccion tiene la lista de las piezas que fueron construidas. Pieza es una clase abstracta.



(a) *Completar el diseño e implementar:*

La tarea es completar el diseño considerando que las piezas pueden ser cilindros, esferas o prismas rectangulares.

Luego, completar la implementación de la clase ReporteDeConstruccion siguiendo la especificación que se muestra a continuación:

- *volumenDeMaterial(nombreDeMaterial: String):*
  "Recibe como parámetro un nombre de material (un string, por ejemplo 'Hierro'). Retorna la suma de los volúmenes de todas las piezas hechas en ese material".
- *superficieDeColor(unNombreDeColor: String):*
  "Recibe como parámetro un color (un string, por ejemplo 'Rojo'). Retorna la suma de las superficies externas de todas las piezas pintadas con ese color".

Fórmulas a utilizar:

- *Cilindro:*

  - *Volumen:* $\pi * radio^2 * h$.
  - *Superficie:* $2\pi * radio * h + 2\pi * radio^2$.
- *Esfera:*
  - *Volumen:* $\frac{4}{3}\pi * radio^3$.
  - *Superficie:* $4\pi * radio^2$.
- *Prisma Rectangular:*
  - *Volumen: ladoMayor * ladoMenor * altura.*
  - *Superficie: 2 (ladoMayor * ladoMenor + ladoMayor * altura + ladoMenor * altura).*

- *Para obtener π, se utiliza Math.PI.*
- *Para elevar un número a cualquier potencia, se utiliza Math.pow(numero: double, potencia: double). Ejemplo:* 8^2 *= Math.pow(8, 2).*

(b) *Pruebas automatizadas:*

Implementar los tests (JUnit) que se consideren necesarios.

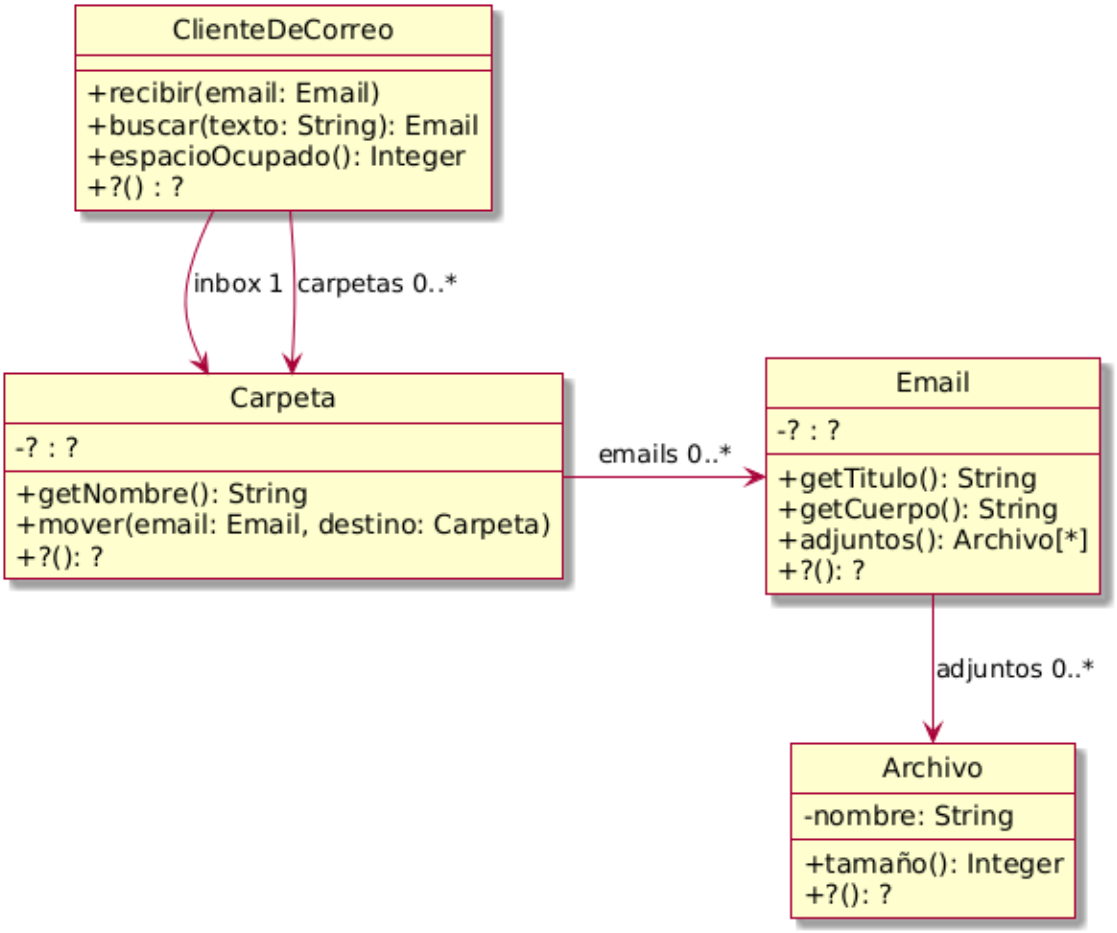
(c) *Discutir con el ayudante:*

Es probable que se note una similitud entre este ejercicio y el de "Figuras y Cuerpos" que se realizó anteriormente, ya que, en ambos, se pueden construir cilindros y prismas rectangulares. Sin embargo, las implementaciones varían. Enumerar las diferencias y similitudes entre ambos ejercicios y, luego, consultar con el ayudante.

Ver proyecto "OO1_E14" de Java.

Ejercicio 15: Cliente de Correo. (*)

El diagrama de clases de UML que se muestra a continuación documenta parte del diseño simplificado de un cliente de correo electrónico.

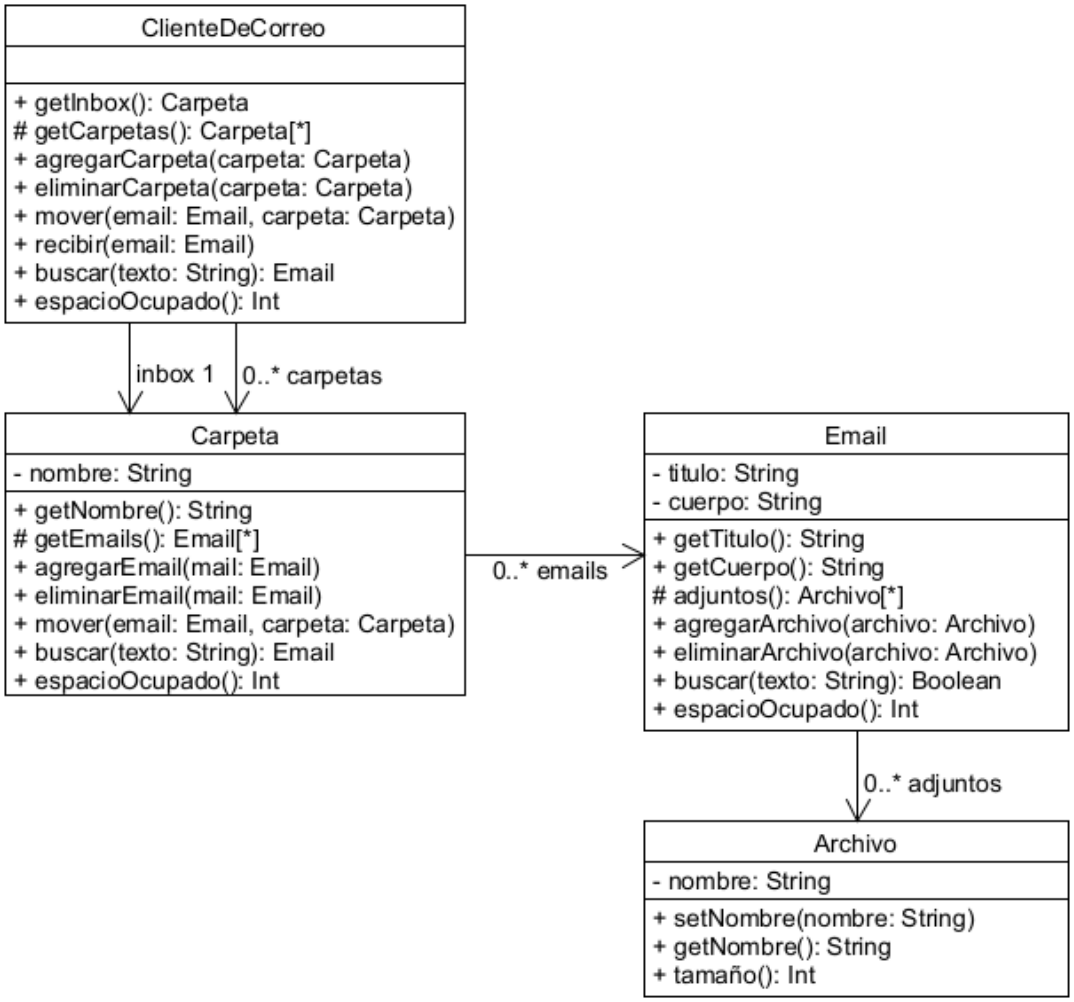


Su funcionamiento es el siguiente:

- *En respuesta al mensaje **recibir**, almacena en el inbox (una de las carpetas) el email que recibe como parámetro.*
- *En respuesta al mensaje **mover**, mueve el email desde una carpeta de origen a una carpeta destino (asumir que el email está en la carpeta origen cuando se recibe este mensaje).*
- *En respuesta al mensaje **buscar**, retorna el primer email en el Cliente de Correo cuyo título o cuerpo contienen el texto indicado como parámetro. Busca en todas las carpetas.*
- *En respuesta al mensaje **espacioOcupado**, retorna la suma del espacio ocupado por todos los emails de todas las carpetas.*
- *El tamaño de un email es la suma del largo del título, el largo del cuerpo y del tamaño de sus adjuntos.*
- *Para simplificar, asumir que el tamaño de un archivo es el largo de su nombre.*

(a) *Modelar e implementar:*

(i) *Completar el diseño y el diagrama de clases UML.*



(ii) *Implementar, en Java, la funcionalidad requerida.*

Ver proyecto “OO1_E15” de Java.

(b) *Pruebas automatizadas:*

> **(i)** *Diseñar los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.*
> **(ii)** *Implementar, utilizando JUnit, los tests automatizados diseñados en el inciso anterior.*

Ver proyecto “OO1_E15” de Java.

Ejercicio 16: Intervalo de Tiempo.

En Java, las fechas se representan, normalmente, con instancias de la clase java.time.LocalDate. Se pueden crear con varios métodos "static" como, por ejemplo, LocalDate.now().

- *Investigar cómo hacer para crear una fecha determinada, por ejemplo 15/09/1972.*
- *Investigar cómo hacer para determinar si la fecha de hoy se encuentra entre las fechas 15/12/1972 y 15/12/2032. Sugerencia: Ver los métodos que permiten que comparar LocalDates y que retornan booleans.*
- *Investigar cómo hacer para calcular el número de días entre dos fechas. Lo mismo para el número de meses y de años Sugerencia: Ver el método until. Tener en cuenta que los métodos de LocalDate colaboran con otros objetos que están definidos a partir de enums, clases e interfaces de java.time; por ejemplo, java.time.temporal.ChronoUnit.DAYS.*

(a) *Implementar:*

Implementar la clase DateLapse (Lapso de tiempo). Un objeto DateLapse representa el lapso de tiempo entre dos fechas determinadas. La primera fecha se conoce como "from" y la segunda como "to". Una instancia de esta clase entiende los mensajes:

(Ver Trabajos Prácticos (P).pdf).

(b) *Pruebas automatizadas:*

> **(i)** *Diseñar los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.*
> **(ii)** *Implementar, utilizando JUnit, los tests automatizados diseñados en el inciso anterior.*

Ver proyecto "OO1_E16" de Java.

Ejercicio 17: Intervalo de Tiempo (¡Otra Vez!).

Asumiendo que se implementó la clase DateLapse con dos variables de instancia "from" y "to", realizar otra implementación de la clase para que la representación sea a través de los atributos "from" y "sizeInDays" y colocarla en otro paquete. Es decir, se debe basar la nueva implementación en estas variables de instancia solamente.

Los cambios en la estructura interna de un objeto sólo deben afectar a la implementación de los métodos. Estos cambios deben ser transparentes para quien envía mensajes, no se debe notar ningún cambio y seguir usándolo de la misma forma. Tener en cuenta que los tests que se implementaron en el ejercicio anterior deberían pasar sin que se requiera realizar modificaciones. Sólo se deberá actualizar el setup, es decir, la instanciación del intervalo de tiempo.

Sugerencia: Para facilitar el uso de estas clases en los tests, considerar definir una interfaz Java que ambas soluciones implementen.

Ver proyecto "OO1_E17" de Java.

Ejercicio 18: *Filtered Set.*

En el apunte de uso de colecciones, se detalla el protocolo de la interface Collection y se ejemplifica el uso mediante la clase concreta ArrayList. Existen otras implementaciones de Collection que tienen ciertas diferencias; en particular, el Set (java.util.Set) es una colección que no admite duplicados y no tiene un índice para acceder a sus elementos.

Implementar una clase EvenNumberSet (conjunto de números pares). Esta especialización se diferencia en que, únicamente, permite agregar números enteros que sean pares. Por simplicidad, considerar, únicamente, el tipo de datos Integer para la solución (ignorar el resto de tipos de datos numéricos).

Tener en cuenta que la clase EvenNumberSet debe implementar la interface Set<Integer> de Java. Esto significa que a las variables de tipo Set<Integer> se les puede asignar un objeto concreto de tipo EvenNumberSet y, luego, utilizarlo enviando los mensajes que están definidos en el protocolo de Set<Integer>.

El siguiente fragmento de código ejemplifica cómo se podría usar la clase EvenNumberSet:

(Ver Trabajos Prácticos (P).pdf).

Evaluar las distintas opciones para implementar la clase EvenNumberSet. Para evitar reinventar la rueda, considerar reutilizar alguna de las clases existentes en Java que ofrezcan funcionalidades similares.

(a) *Investigar qué clases se pueden utilizar para implementar la clase EvenNumberSet. Consultar la documentación de Set.*

(b) *Explicar, brevemente, cómo se proponen utilizar las clases investigadas anteriormente para implementar la solución. Por ejemplo:*

- *"Se debe subclasificar una determinada clase y redefinir un método para que haga lo siguiente".*
- *"Se debe crear una nueva clase que contenga un objeto de un determinado tipo al cual se le delegará está responsabilidad".*

(c) *Implementar, en Java, las alternativas que se hayan propuesto.*

(d) *Implementar tests automatizados, utilizando JUnit, para verificar las implementaciones.*

(e) *Comparar las soluciones y listar las ventajas y desventajas de cada una.*

Ver proyecto "OO1_E18" de Java.

Ejercicio 19: Alquiler de Propiedades. (*)

Ejercicio 20: Políticas de Cancelación.

Ejercicio 21: Servicio de Envíos de Paquetes. (*)

Ejercicio 22: Sistema de Pedidos. (*)

Se desea diseñar un sistema para una empresa que ofrece viandas preparadas y artículos de supermercado. De las viandas, se conoce su nombre, precio, si es apta para celíacos y disponibilidad. De los artículos de supermercado, se conoce su nombre, categoría, precio, peso en kg. y stock actual.

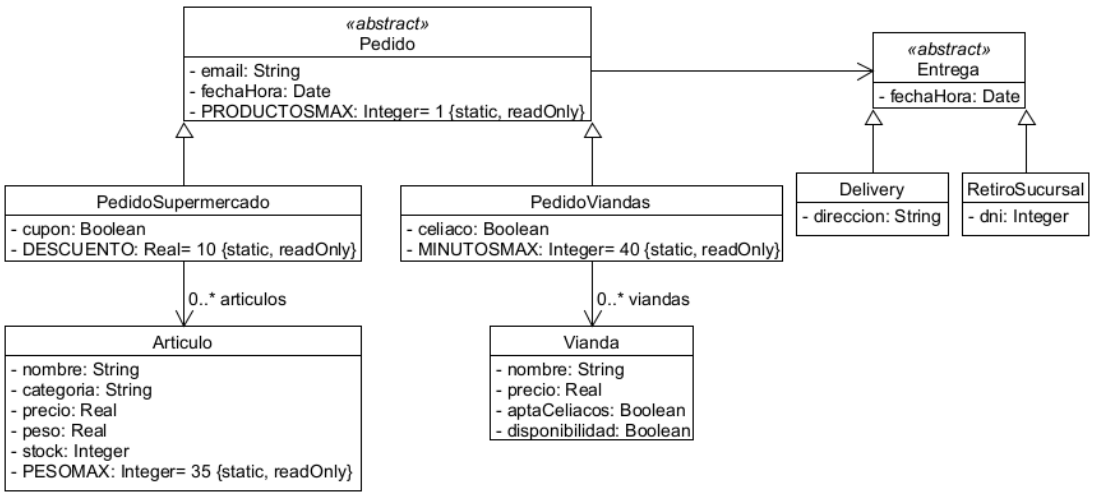
Para realizar un pedido, no es necesario registrarse en la aplicación, alcanza con indicar el email al momento de solicitar los productos. Por cuestiones logísticas, cada pedido deberá ser de uno de los dos rubros: supermercado o viandas. Para coordinar la entrega, se puede optar por envío por delivery, en ese caso se debe indicar la dirección y fecha/hora en la que se recibirá el pedido; o bien, retiro en sucursal, para lo cual se deberá indicar el DNI de la persona autorizada a retirar y la fecha/hora estimadas en la que retira.

La empresa dispone de cocina sin TACC, por lo que es muy importante al momento de realizar un pedido que se indique sí es apto celíaco o no. Los pedidos de supermercado pueden obtener un 10% de descuento en caso de que se utilice un cupón de descuento al solicitarlos.

Se quiere que el sistema permita las siguientes funcionalidades:

- *Crear pedido: Se indica el tipo de pedido, los productos solicitados (por simplicidad, se puede pedir sólo una unidad de cada uno) y la forma de entrega. En caso de solicitar viandas, debe indicar sí es apto celíaco. En caso de solicitar artículos de supermercado, se puede incluir un cupón de descuento.*
- *Calcular costo pedido: Dado un pedido, se calcula el costo sumando todos los precios de los productos que contenga el pedido.*
- *Calcular peso pedido: Dado un pedido de supermercado, se calcula el peso sumando todos los pesos de los productos que contenga el pedido.*
- *Calcular tiempo estimado de entrega: Dado un pedido y de acuerdo a la forma de entrega, se calculan los minutos estimados hasta la entrega de la siguiente manera:*
  - *Retiro: Es la diferencia entre la fecha/hora de creación del pedido y la fecha/hora estimada de retiro.*
  - *Envío por delivery: La empresa lo calcula, en tiempo real, utilizando GPS, el estado del tráfico y fórmulas de distancia.*
- *Agregar artículo a pedido: Agrega el artículo al pedido siempre y cuando su tipo sea supermercado y no supere el peso máximo de 35 kg.*
- *Agregar vianda a pedido: Agrega la vianda al pedido siempre y cuando su tipo sea de viandas y falten menos de 40 min estimados para su entrega; si el pedido es sólo apto celíaco, sólo admitirá viandas apto celíaco.*

Realizar un diagrama de clases UML donde se identifiquen todas las clases del dominio, sus atributos, roles, cardinalidades y asociaciones. No se debe incluir comportamiento.


«abstract»
Pedido
- email: String
- fechaHora: Date
- PRODUCTOSMAX: Integer= 1 {static, readOnly}
«abstract»
Entrega
- fechaHora: Date
PedidoSupermercado
- cupon: Boolean
- DESCUENTO: Real= 10 {static, readOnly}
PedidoViandas
- celiaco: Boolean
- MINUTOSMAX: Integer= 40 {static, readOnly}
Delivery
- direccion: String
RetiroSucursal
- dni: Integer
0..* articulos
0..* viandas
Articulo
- nombre: String
- categoria: String
- precio: Real
- peso: Real
- stock: Integer
- PESOMAX: Integer= 35 {static, readOnly}
Vianda
- nombre: String
- precio: Real
- aptaCeliacos: Boolean
- disponibilidad: Boolean

Ejercicio 23: *PoolCar*. (*)

PoolCar es una aplicación para organizar viajes de larga distancia, que permite compartir gastos y abaratar el costo del viaje.

De cada usuario, se conoce su nombre y dirección. Además, se mantiene un estado de cuenta, que se usa para el pago de sus viajes. Al momento de su alta, los usuarios se registran como conductor o pasajero, lo cual es una característica que no se puede modificar.

Los conductores deben registrar, al menos, un vehículo. De cada uno, se conoce una descripción, su capacidad (incluyendo al conductor), el año de fabricación y el valor de mercado. Un vehículo pertenece siempre a un único conductor y cada conductor sólo podrá viajar en calidad de chofer de su propio vehículo.

Los viajes son creados por los conductores y quedan asociados a uno de sus vehículos. De cada viaje, se registra el costo total, la localidad de origen, la localidad de destino y la fecha de salida.

Los pasajeros pueden inscribirse en los viajes creados por conductores, siempre y cuando haya lugares disponibles en el vehículo y su saldo de cuenta sea positivo. La inscripción sólo está permitida hasta dos días antes de la fecha de salida del viaje.

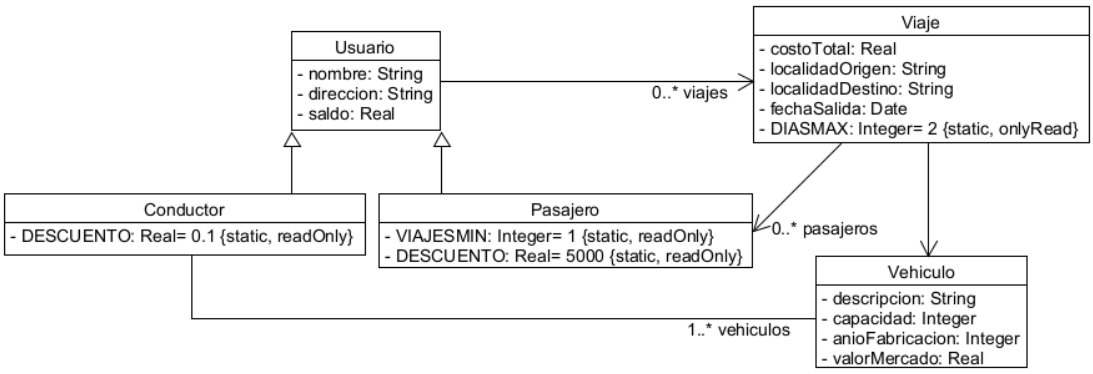
Cuando un viaje se lleva a cabo, el costo total del mismo se divide en partes iguales entre todos los participantes, tanto pasajeros como conductor, descontándose de sus respectivos saldos de cuenta. Además, se aplican descuentos:

- *Para el conductor, se descuenta un monto equivalente al 0,1% del valor de su vehículo.*
- *Para cada pasajero que ya haya realizado, al menos, un viaje previamente, se descuenta un monto fijo de $5.000.*

El sistema debe permitir:

- *Cargar saldo para la cuenta del usuario: Dado un usuario y un monto, se carga ese monto en el saldo de cuenta del usuario.*
- *Crear un viaje desde un origen a un destino: Dado un chofer y un vehículo de su propiedad, crear un viaje a un destino, indicando la fecha de salida.*
- *Registrar un pasajero para un viaje: Dado un viaje, y un pasajero, se registra el pasajero en el viaje indicado.*
- *Cobrar a cada participante el costo de un viaje: Dado un viaje, descontar, del saldo de cada participante, el costo que le corresponde por el mismo. El saldo de la cuenta puede quedar en negativo.*
- *Listar los destinos visitados por un usuario: Dado un usuario, se quiere obtener la lista de destinos visitados.*

Realizar un diagrama de clases UML donde se identifiquen todas las clases del dominio, sus atributos, roles, cardinalidades y asociaciones. No se debe incluir comportamiento.


Usuario
- nombre: String
- direccion: String
- saldo: Real
Viaje
- costoTotal: Real
- localidadOrigen: String
- localidadDestino: String
- fechaSalida: Date
- DIASMAX: Integer= 2 {static, onlyRead}
0..* viajes
Conductor
- DESCUENTO: Real= 0.1 {static, readOnly}
Pasajero
- VIAJESMIN: Integer= 1 {static, readOnly}
- DESCUENTO: Real= 5000 {static, readOnly}
0..* pasajeros
Vehiculo
- descripcion: String
- capacidad: Integer
- anioFabricacion: Integer
- valorMercado: Real
1..* vehiculos

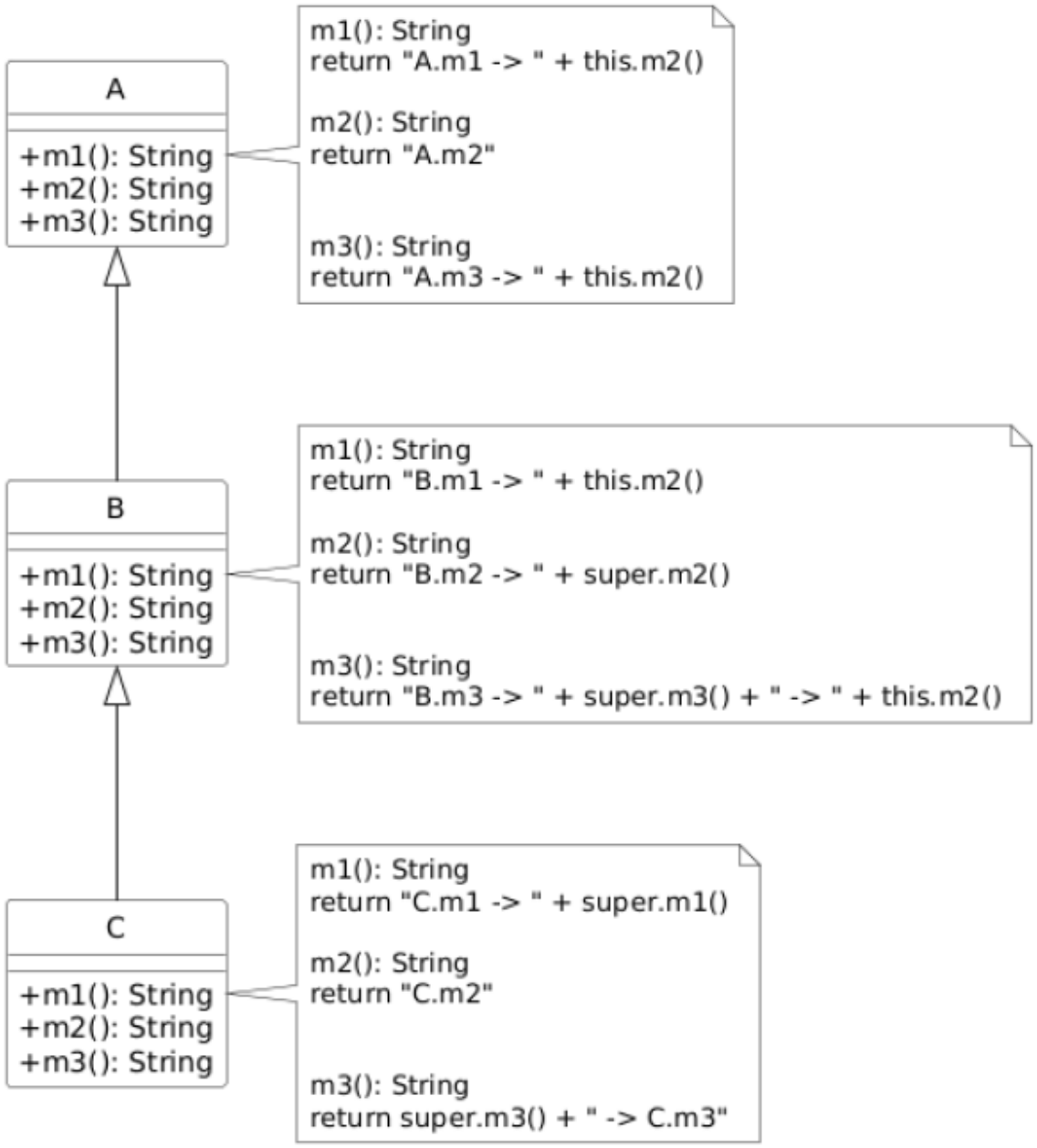
Ejercicio 24: *GreenHOOme*.

Ejercicio 25: *Bag*.

Ejercicio 26: Estadísticas del Cliente de Correo. (*)

Ejercicio 27: *Method Lookup*.

Dado el siguiente modelo y los fragmentos de códigos mostrados, marcar la respuesta para cada una de las preguntas enunciadas. Sólo una opción es la correcta.



(a) *Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:*

```
C c= new C();
c.m1();
```

1. C.m1 -> B.m1 -> C.m2.
2. C.m1 -> A.m1 -> A.m2.
3. C.m1 -> B.m1 -> B.m2 -> A.m2.
4. C.m1 -> A.m1 -> C.m2.

Ver proyecto “OO1_E27” de Java.

(b) *Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:*

```
C c= new C();
c.m2();
```

1. A.m2.
2. B.m2 -> A.m2.
3. C.m2.
4. B.m2 -> C.m2.

Ver proyecto “OO1_E27” de Java.

(c) *Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:*

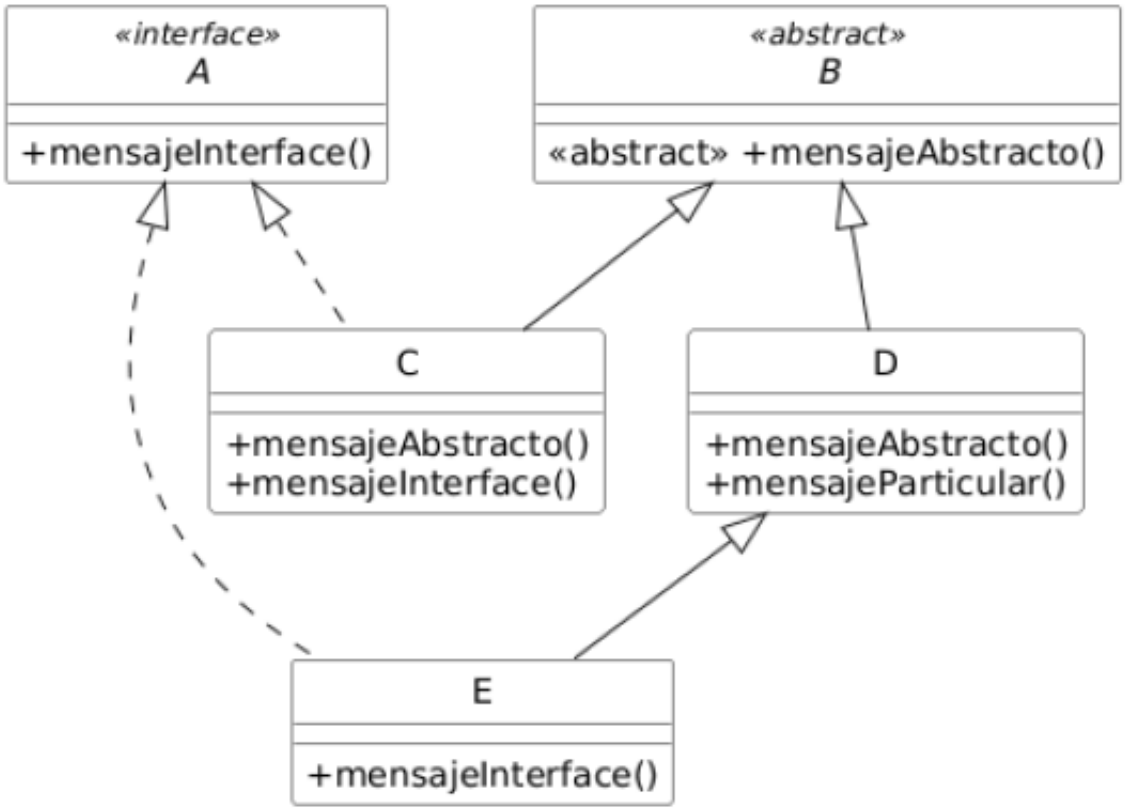
```
C c= new C();
c.m3();
```

1. B.m3 -> A.m3 -> C.m2 -> C.m2.
2. B.m3 -> A.m3 -> C.m2 -> C.m2 -> C.m3.
3. B.m3 -> B.m3 -> A.m2 -> A.m2 -> C.m3.
4. B.m3 -> A.m3 -> C.m2 -> C.m3.

Ver proyecto “OO1_E27” de Java.

Ejercicio 28: *TipadOOs*.

Dado el diagrama de clases UML proporcionado, completar todos los bloques de código reemplazando los signos de interrogación (???) con los tipos y métodos correctos. Escribir todas las combinaciones válidas posibles.



(a)

A objeto= new ???();
objeto.???;

A objeto= new C();
objeto.mensajeAbstracto;
objeto.mensajeInterface;

A objeto= new E();
objeto.mensajeAbstracto;
objeto.mensajeParticular;
objeto.mensajeInterface;

Ver proyecto “OO1_E28” de Java.

(b)

B objeto= new ???();

objeto.???;

B objeto= new C();
objeto.mensajeAbstracto;
objeto.mensajeInterface;

B objeto= new D();
objeto.mensajeAbstracto;
objeto.mensajeParticular;

B objeto= new E();
objeto.mensajeAbstracto;
objeto.mensajeParticular;
objeto.mensajeInterface;

Ver proyecto “OO1_E28” de Java.

(c)

D objeto = new ???();
objeto.???;

D objeto= new D();
objeto.mensajeAbstracto;
objeto.mensajeParticular;

D objeto= new E();
objeto.mensajeAbstracto;
objeto.mensajeParticular;
objeto.mensajeInterface;

Ver proyecto “OO1_E28” de Java.

(d)

C objeto= new C();
objeto.???;

C objeto= new C();
objeto.mensajeAbstracto;
objeto.mensajeInterface;

Ver proyecto “OO1_E28” de Java.

(e)

??? objeto= new C();
objeto.mensajeAbstracto;

B objeto= new C();
objeto.mensajeAbstracto;

C objeto= new C();
objeto.mensajeAbstracto;

Ver proyecto “OO1_E28” de Java.

(f)

??? objeto= new C();
objeto.mensajeInterface;

A objeto= new C();
objeto.mensajeInterface;

C objeto= new C();
objeto.mensajeInterface;

Ver proyecto “OO1_E28” de Java.

Ejercicio 29: Plataforma de *Streaming*. (*)