Cuadernillo Semestral de Actividades

Actualizado: 17 de octubre de 2023

El presente cuadernillo posee un compilado con todos los ejercicios que se usarán durante el semestre en la asignatura. Los ejercicios están organizados en forma secuencial, siguiendo los contenidos que se van viendo en la materia.

Cada semana les indicaremos cuales son los ejercicios en los que deberían enfocarse para estar al día y algunos de ellos serán discutidos en la explicación de práctica.

Recomendación importante:
Los contenidos de la materia se incorporan y fijan mejor cuando uno intenta aplicarlos - **no alcanza con ver un ejercicio resuelto por alguien más**. Para sacar el máximo provecho de los ejercicios, es importante que asistan a las consultas de práctica habiendo intentado resolverlos (tanto como les sea posible). De esa manera podrán hacer consultas más enfocadas y el docente podrá darles mejor feedback.

Ejercicio 1: WallPost

Primera parte

Se está construyendo una red social como Facebook o Twitter. Debemos 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, utilice el recurso que se encuentra en el sitio de la cátedra. Para importar el proyecto, siga los pasos explicados en el documento “*Trabajando con proyectos Maven*, *importar un proyecto”*. Allí verá que existe la interface Wallpost y la clase WallpostImpl que implementa la interfaz anterior. Una vez importado, dentro del mismo, debe completar la clase WallPostImpl para que entienda:

```
/*
* Permite construir una instancia del WallpostImpl.
* Luego de la invocación, debe tener como texto: “Undefined post”,
* no debe estar marcado como destacado y la cantidad de “Me gusta” debe ser 0.
*/
public WallPostImpl()
```

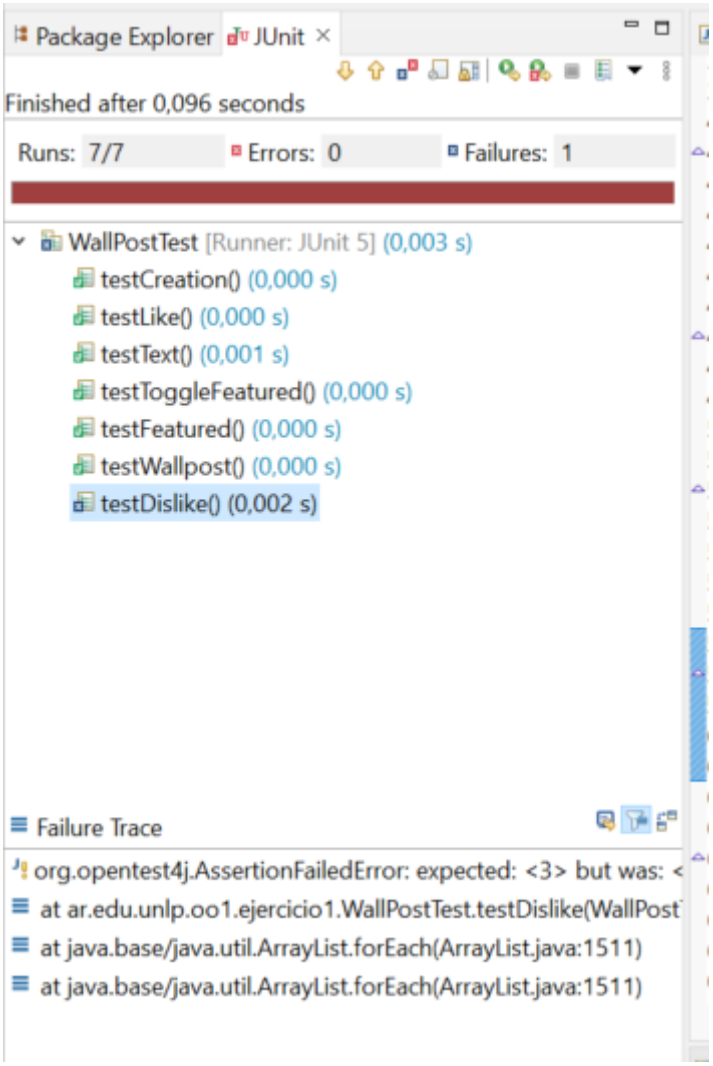
E implemente el protocolo definido en la interfaz Wallpost como se detalla a continuación

```
/*
* Retorna el texto descriptivo de la publicación
*/
public String getText()

/*
* Asigna el texto descriptivo de la publicación
*/
public void setText (String descriptionText)

/*
* Retorna la cantidad de “me gusta”
*/
public int getLikes()

/*
 * Incrementa la cantidad de likes en uno.
*/
public void like()

/*
 * Decrementa la cantidad de likes en uno. Si ya es 0, no hace nada.
*/
public void dislike()

/*
 * Retorna true si el post está marcado como destacado, false en caso contrario
*/
public boolean isFeatured()

/*
 * Cambia el post del estado destacado a no destacado y viceversa.
*/
public void toggleFeatured()
```

Segunda parte

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

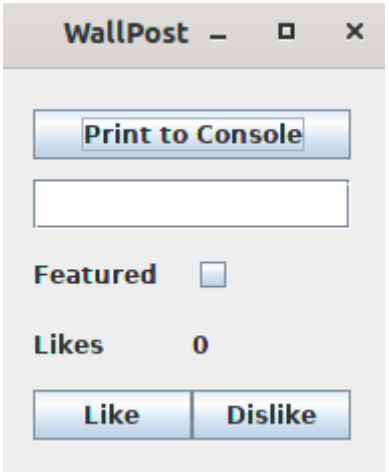
Para ejecutar los tests simplemente haga click derecho sobre el proyecto y utilice la opción Run As >> JUnit Test. Al ejecutarlo, se abrirá una ventana con el resultado de la evaluación de los tests. Siéntase libre de investigar la implementación de la clase de test. Ya veremos en detalle cómo implementarlas.



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 lo selecciona, mostrará el mensaje de error correspondiente a ese test, que le ayudará a encontrar la falla. Si hace click sobre alguno de los test, se abrirá su implementación en el editor.

Tercera parte

Una vez que su implementación pasa los tests de la primera parte 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.

Ejercicio 2: Balanza Electrónica

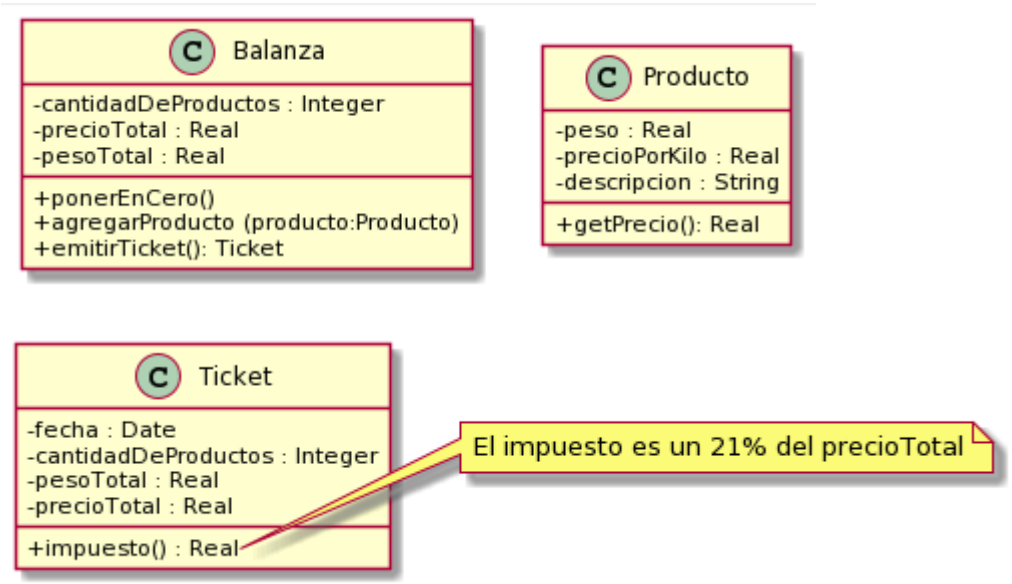
En el taller de programación ud programó una balanza electrónica. Volveremos a programarla, con algún requerimiento adicional.

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.

Implemente:

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



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

Todas las clases son subclases de Object.

Nota: Para las fechas, utilizaremos la clase java.time.LocalDate. Para crear la fecha actual, puede utilizar LocalDate.now(). También es posible crear fechas distintas a la actual. Puede investigar más sobre esta clase en https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html

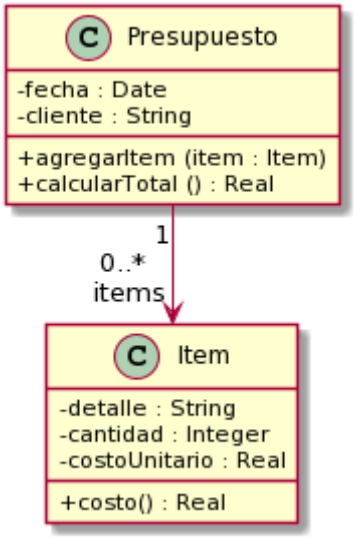
Probando su implementación:

Para realizar este ejercicio, utilice el recurso que se encuentra en el sitio de la cátedra. En este caso, se trata de dos clases, BalanzaTest y ProductoTest, las cuales debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores.

Si todo salió bien, su 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.

Ejercicio 3: Presupuestos

Defina el proyecto Ejercicio 3 - Presupuesto y dentro de él implemente las clases que se observan en el siguiente diagrama. Ambas son subclases de Object. Preste atención a los siguientes aspectos:



- ¿Cuáles son las variables de instancia de cada clase?
- ¿Qué variables inicializa y cómo?

Probando su código:
Utilice los tests provistos para confirmar que su implementación ofrece la funcionalidad esperada. En este caso, se trata de dos clases, ItemTest y PresupuestoTest, que debe agregar

dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores. Siéntase libre de explorar las clases de test para intentar entender qué es lo que hacen.

Ejercicio 3 - Bis: Balanza mejorada

Realizando el ejercicio de los presupuestos, aprendimos que un objeto puede tener una colección de otros objetos. Con esto en mente, ahora queremos mejorar la balanza implementada anteriormente.

Tarea 1

Mejorar la balanza para que recuerde los productos ingresados (los mantenga en una colección). Analice de qué forma puede realizarse este nuevo requerimiento e implemente 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 el mensaje *ponerEnCero()* ?

¿Es necesario, ahora, almacenar los totales en la balanza? ¿Se pueden obtener estos valores de otra forma?

Tarea 2

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

¿Qué cambios cree necesarios en Ticket para que pueda conocer a los productos?

Tarea 3

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

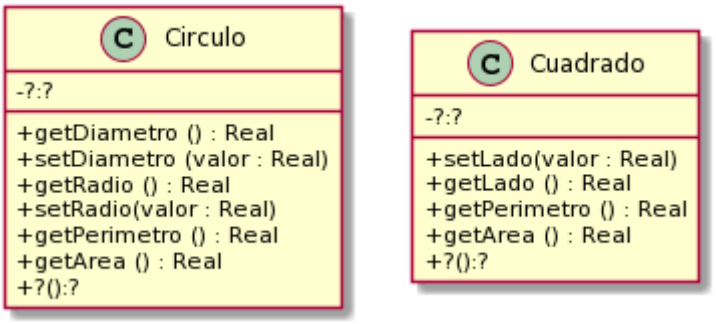
Ejercicio 4: Figuras y cuerpos

Figuras en 2D

Defina un nuevo proyecto figurasYCuerpos

En Taller de Programación definió clases para representar figuras geométricas. Retomaremos ese ejercicio para trabajar con Cuadrados y Círculos.

El siguiente diagrama de clases documenta los mensajes que estos objetos deben entender. Decida usted qué variables de instancia son necesarias. Ambas clases son subclases de Object. Puede agregar mensajes adicionales si lo cree necesario.

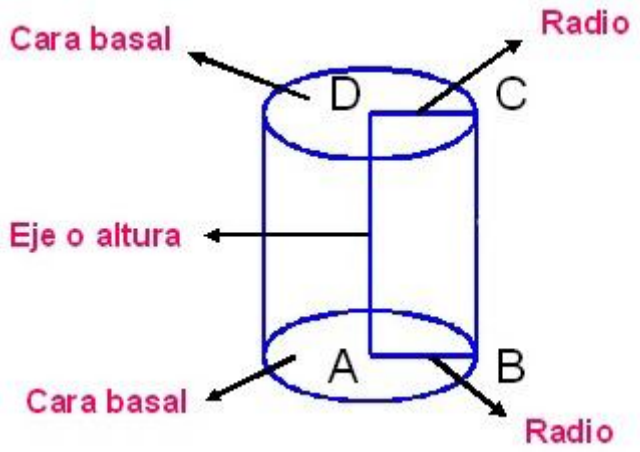


Fórmulas y mensajes útiles:

- 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)

Cuerpos en 3D

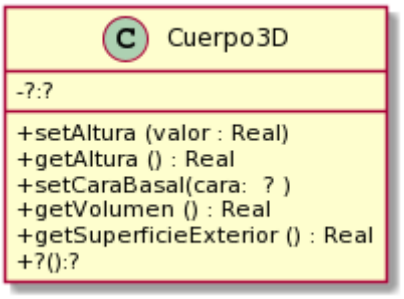
Ahora que tenemos Círculos y Cuadrados, podemos usarlos para construir cuerpos (en 3D) y calcular su volumen y superficie o área exterior. Vamos a pensar a un cilindro como "un cuerpo que tiene una figura 2D como cara basal y que tiene una altura (vea la siguiente imagen)" . Si en el lugar de la figura2D tuviera un círculo, se formaría el siguiente cuerpo 3D.



Si reemplazamos la cara basal por un rectángulo, tendremos un prisma (una caja de zapatos).

El siguiente diagrama de clases documenta los mensajes que entiende un cuerpo3D. Decida usted qué variables de instancia son necesarias. Cuerpo3D es subclase de Object.

Decida usted si es necesario hacer cambios en las figuras 2D.



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 llama generatriz. En wikipedia (Cilindro)[1] se puede aprender un poco más al respecto.

Pruebas automatizadas

Siguiendo los ejemplos de ejercicios anteriores, ejecute las pruebas automatizadas provistas. En este caso, se trata de tres clases que debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consulte al ayudante.

Discuta y reflexione

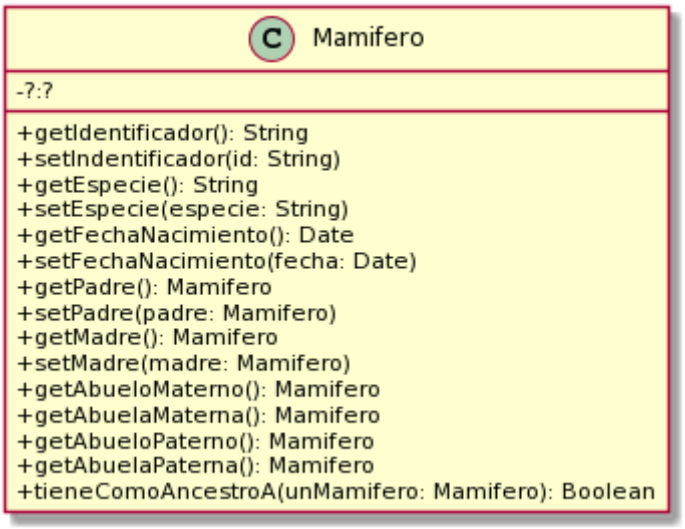
Discuta con el ayudante sus elecciones de variables de instancia y métodos adicionales. ¿Es necesario todo lo que definió?

Ejercicio 5: 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 nos han pedido ayuda. Debemos:

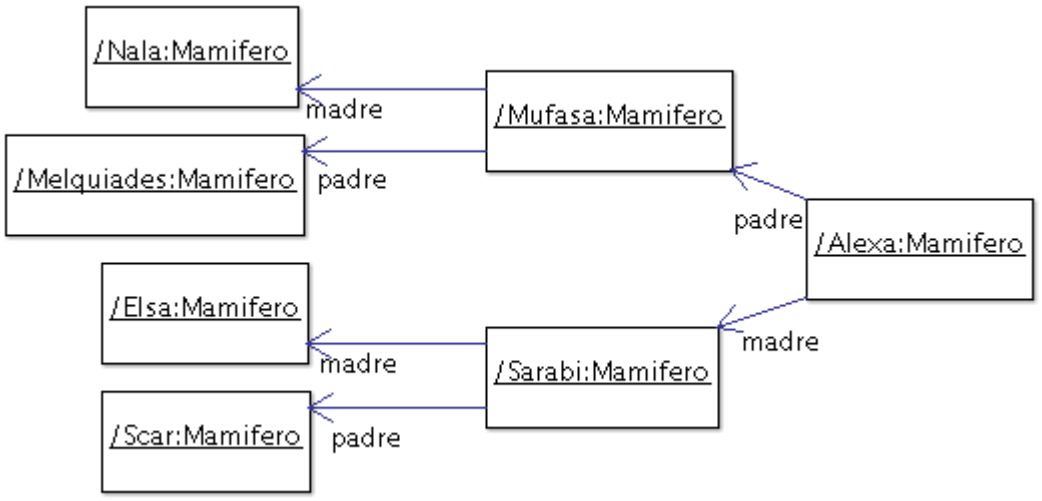
a) Modelar en objetos y programar la clase Mamífero (como subclase de Object). El siguiente diagrama de clases (incompleto) nos da una idea de los mensajes que un mamífero entiende.
Deje tieneComoAncestroA para el final y discuta su solución con el ayudante.

[1] https://es.wikipedia.org/wiki/Cilindro



b) Complete el diagrama de clases para reflejar los atributos y relaciones requeridos.

c) Siguiendo los ejemplos de ejercicios anteriores, ejecute las pruebas automatizadas provistas. En este caso, se trata de una clase, MamiferoTest, que 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).

Haga las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consulte al ayudante.

Ejercicio 6: Red de Alumbrado

Imagine una red de alumbrado donde cada farola está conectada a una o varias vecinas formando un grafo conexo[2]. 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 bi-direccionales)
3. encender una farola (y obtener el efecto antes descrito)
4. apagar una farola (y obtener el efecto antes descrito)

Tareas:

1. Realice el diagrama UML de clases de la solución al problema.

2. Implemente en Java, la clase Farola, como subclase de Object, con los siguientes métodos:

```
/*
* Crear una farola. Debe inicializarla como apagada
*/
public Farola ()
/*
* Crea la relación de vecinos entre las farolas. La relación de vecinos
entre las farolas es recíproca, es decir el receptor del mensaje será vecino
de otraFarola, al igual que otraFarola también se convertirá en vecina del
receptor del mensaje
*/
public void pairWithNeighbor( Farola otraFarola )
/*
* Retorna sus farolas vecinas
*/
public List<Farola> getNeighbors ()

/*
* Si la farola no está encendida, la enciende y propaga la acción.
*/
public void turnOn()

/*
* Si la farola no está apagada, la apaga y propaga la acción.
*/
public void turnOff()
```

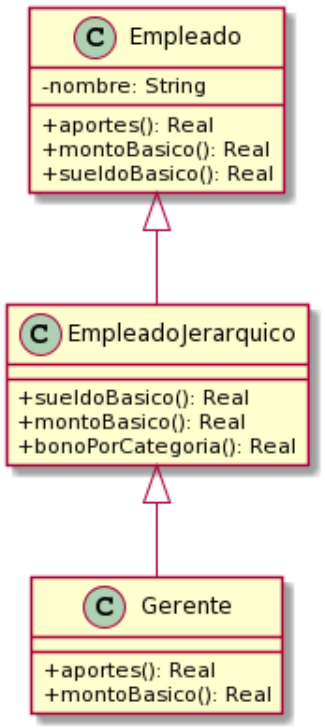
[2] https://es.wikipedia.org/wiki/Grafo_conexo

```
/*
* Retorna true si la farola está encendida.
*/
public boolean isOn()
```

3. Utilice los tests provistos por la cátedra para probar las implementaciones del punto 2.

Ejercicio 7: 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 de la derecha.



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; }`	

Analice cada uno de los siguientes fragmentos de código y resuelva las tareas indicadas abajo:

Gerente alan = **new** Gerente("Alan Turing"); **double** aportesDeAlan = alan.aportes();	Gerente alan = **new** Gerente("Alan Turing"); **double** sueldoBasicoDeAlan = alan.sueldoBasico();

Tareas

1. Liste los métodos que son ejecutados como resultado del envío del último mensaje (por ejemplo, método #aportes de la clase X, ...)
2. Responda qué valores tendrán las variables aportesDeAlan y sueldoBasicoDeAlan.

Ejercicio 8: Distribuidora Eléctrica

Una distribuidora eléctrica desea un sistema para el registro de los consumos de sus usuarios y para la emisión de facturas de cobro.

El sistema permite registrar usuarios, para los cuales se indica nombre y dirección. Por simplificación, un usuario puede estar relacionado con un solo domicilio (para el que se registran los consumos).

El sistema permite registrar los consumos para los usuarios. Los consumos que se registran para los usuarios tienen dos componentes: el consumo de energía activa y el consumo de energía reactiva.

Una vez al mes, la empresa distribuidora realiza el proceso de facturación. Por cada usuario se emite una factura (el proceso completo retorna una colección).

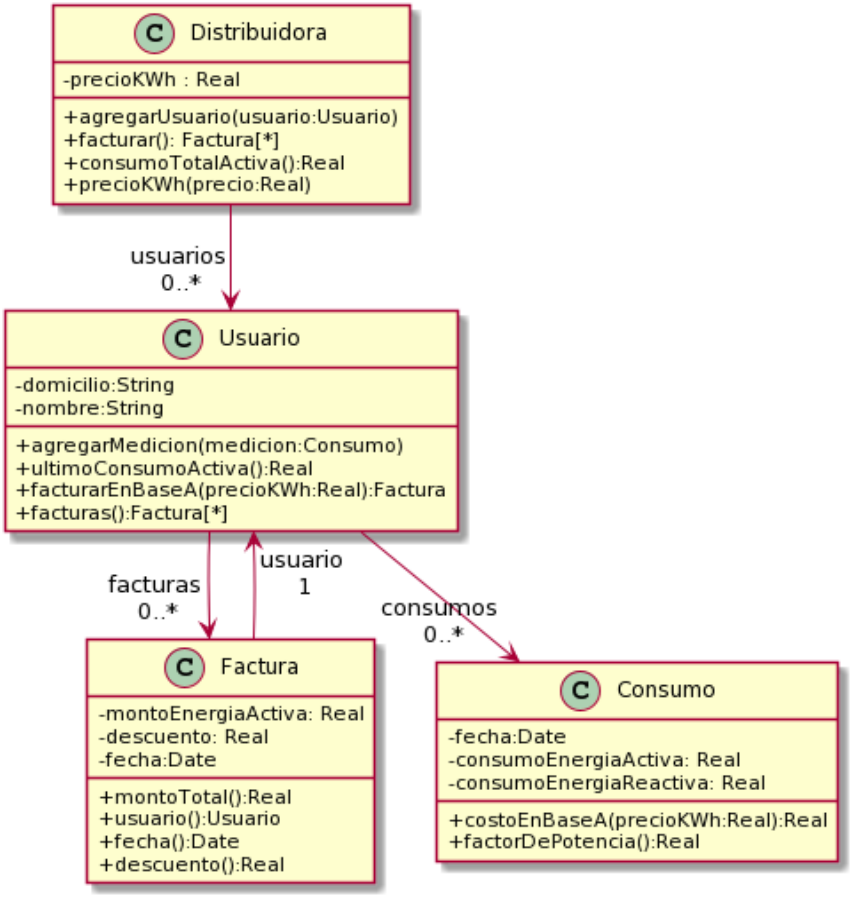
Para emitir la factura de un cliente se tiene en cuenta su último consumo y se calcula su factor de potencia para determinar si hay alguna bonificación para aplicar. El costo del consumo se calcula multiplicando el consumo de energía activa por el precio del kwh (Kilowatt/hora) de la empresa. La energía reactiva no tiene costo para el usuario. Si el factor de potencia estimado (pfe) del último consumo del usuario es mayor a 0.8, el usuario es bonificado con el 10%.

El factor de potencia se calcula de acuerdo a la siguiente fórmula:

$$fpe = \frac{EnergiaActiva}{\sqrt{EnergiaActiva^2 + EnergiaReactiva^2}}$$

Además, la empresa está interesada en poder saber cuál fue el total de energía activa consumida por toda la red en el último periodo medido (es decir, teniendo en cuenta sólo la última medición de cada usuario).

El siguiente diagrama de clases muestra el diseño para este problema. Agregue los métodos que considere necesarios.



Tareas

Siguiendo el diseño que se muestra en el diagrama de clases, implemente la funcionalidad que se describe en el enunciado, en particular en lo referente a:

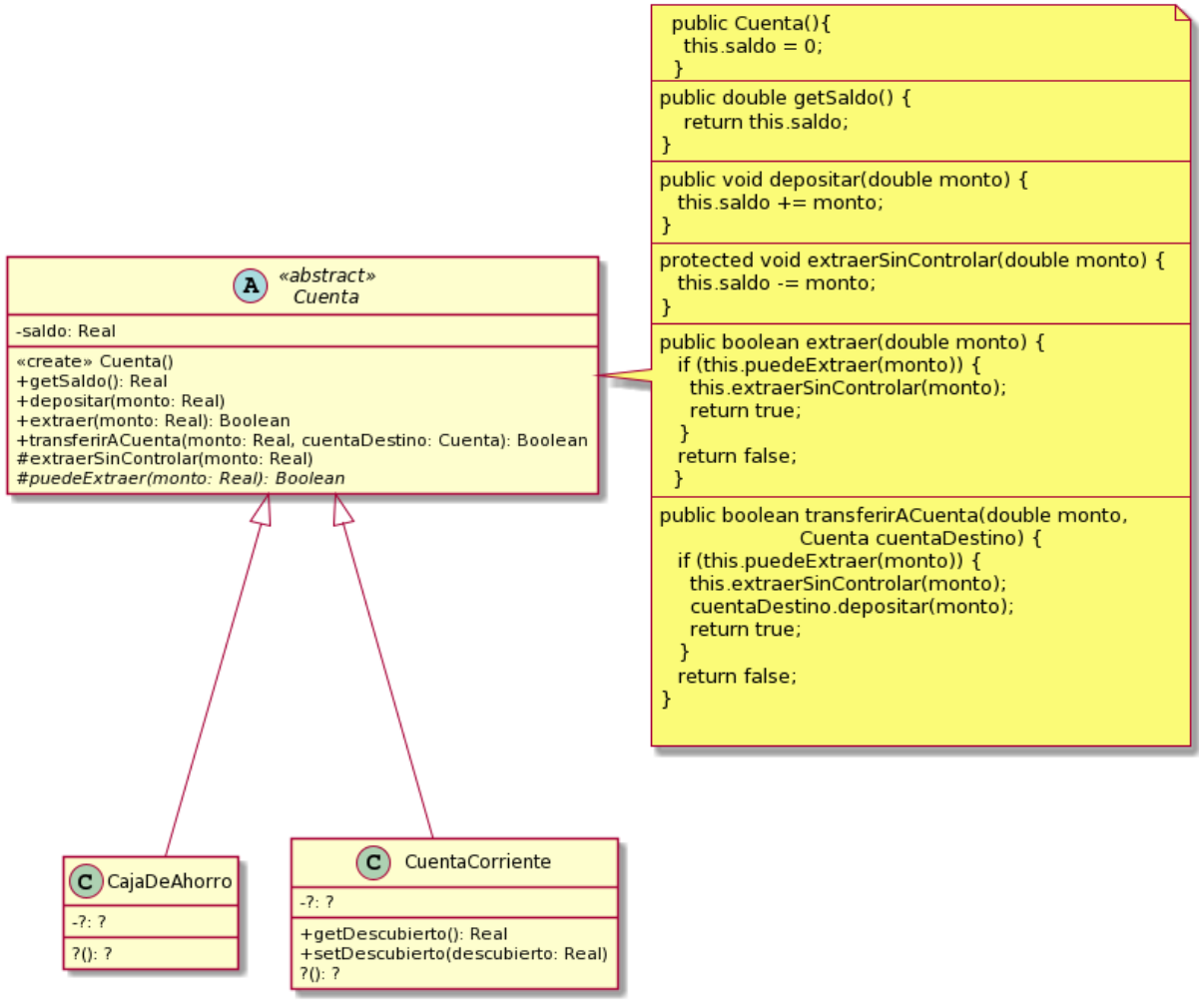
1. Establecer (setear) el precio del KWh de la empresa
2. Agregar usuarios
3. Agregar mediciones
4. Emitir facturas
5. Obtener el consumo total en KWh de la red para el último período

En una clase de test, cree los test de unidad necesarios para poder evaluar:

1. Iniciar el sistema
2. Agregar un usuario
3. Agregar un consumo para ese usuario
4. Emitir las facturas
5. Calcular el consumo total de la red para el último período

Ejercicio 9 : Cuenta con ganchos

Observe 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.



Tarea A: Complete la implementación de las clases `CajaDeAhorro` y `CuentaCorriente` para que se puedan efectuar depósitos, extracciones y transferencias teniendo en cuenta los siguientes criterios.

1) Las **cajas de ahorro** *solo pueden extraer y transferir* cuando cuentan con fondos suficientes.
2) Las extracciones, los depósitos y las transferencias desde **cajas de ahorro** tienen un costo adicional de 2% del monto en cuestión (téngalo en cuenta antes de permitir una extracción o transferencia desde caja de ahorro).
3) Las **cuentas corrientes** pueden extraer aún 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).

4) Cuando se abre una **cuenta corriente**, su límite descubierto es 0 (no olvide definir el constructor por default).

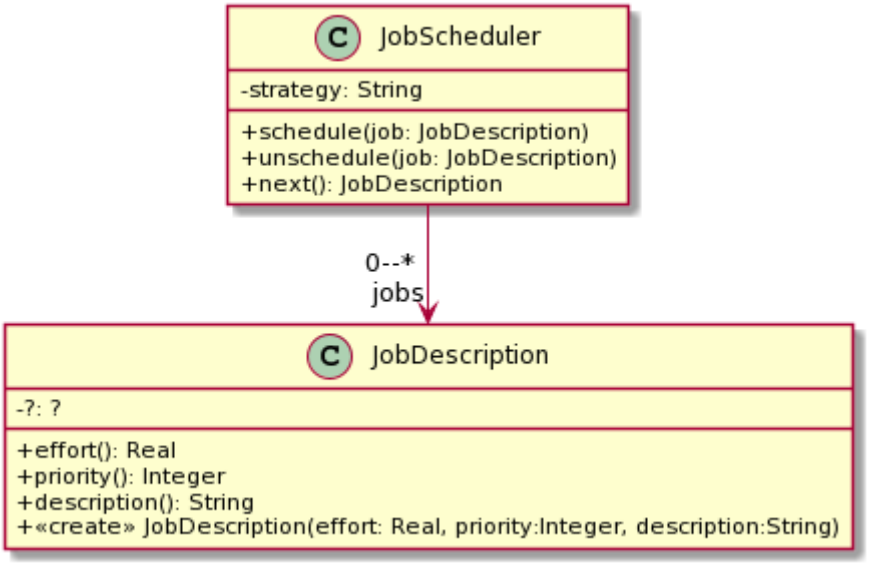
Tarea B: Reflexione, charle con el ayudante y responda a las siguientes preguntas.

a) ¿Por qué cree que este ejercicio se llama "Cuenta con ganchos"?
b) En las implementaciones de los métodos `extraer()` y `transferirACuenta()` que se ven en el diagrama, ¿quién es `this`? ¿Puede decir de qué clase es `this`?
c) ¿Por qué decidimos que los métodos `puedeExtraer()` y `extraerSinControlar` tengan visibilidad "protegido"?
d) ¿Se puede transferir de una caja de ahorro a una cuenta corriente y viceversa? ¿por qué? ¡Pruébelo!
e) ¿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?

Tarea C: Escriba los tests de unidad que crea necesarios para validar que su implementación funciona adecuadamente.

Ejercicio 10 - 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.

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

Sus cambios probablemente hagan que los tests dejen de funcionar. Corríjalos y mejórelos como sea necesario.

Ejercicio 11 - El Inversor

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. Nuestro sistema representa al inversor y a cada uno de los tipos de inversiones con una clase.

- La clase InversionEnAcciones tiene las siguientes variables de instancia:
    String nombre;
    int cantidad;
    double valorUnitario;

- La clase PlazoFijo tiene las siguientes variables de instancia:
    LocalDate fechaDeConstitucion;
    double montoDepositado;
    double porcentajeDeInteresDiario;

- La clase Inversor tiene las siguientes variables de instancia:
    String nombre;
    List<?> inversiones;

La variable inversiones de la clase Inversor es una colección con instancias de cualquiera de las dos clases de inversiones que pueden estar mezcladas.
Cuando se quiere saber cuánto dinero representan las inversiones del inversor, se envía al mismo el mensaje *valorActual()*.

1. Implemente en Java lo que considere necesario para que las instancias de Inversor entiendan el mensaje *valorActual()* teniendo en cuenta los siguientes criterios:
    - el valor actual de las inversiones de un inversor es la suma de los valores actuales de cada una de las inversiones en su cartera (su colección de inversiones).

- el valor actual de un plazo fijo equivale al montoDepositado incrementado como corresponda por el porcentaje de interés diario, desde la fecha de constitución a la fecha actual (la del momento en el que se hace el cálculo).
- el valor actual de una InversionEnAcciones se calcula multiplicando el número de acciones por el valor unitario de las mismas.

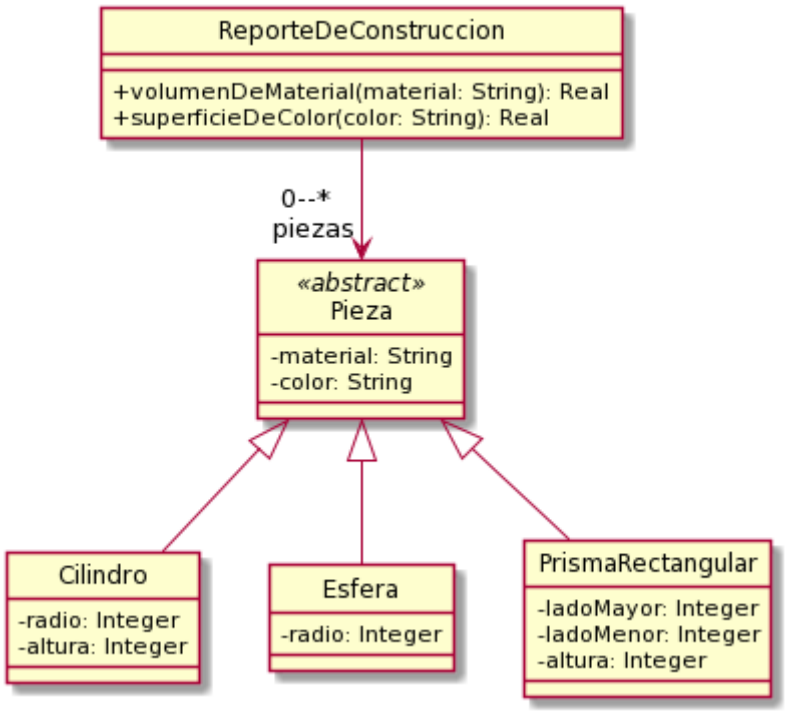
Recordatorio: No olvide la inicialización.

2. Construya un diagrama UML de clases para documentar su solución.
3. Implemente los tests (JUnit) que considere necesarios.

Ejercicio 12: 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.



Tarea

Su tarea es completar el diseño e implementarlo siguiendo las especificaciones que se exponen a continuación:

getVolumenDeMaterial(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"

getSuperficieDeColor(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".

Pruebas de unidad

Asegúrese de proveer tests de unidad para todo el comportamiento desarrollado.

Fórmulas

Volumen de un cilindro: π * radio ^{2} * h.
Superficie de un cilindro: 2 * π * radio * h + 2 * π * radio ^{2}
Volumen de una esfera: ⅓ * π * radio ³.
Superficie de una esfera: 4 * π * radio ^{2}
Volumen del prisma: ladoMayor * ladoMenor * altura
Superficie del prisma: 2 * (ladoMayor * ladoMenor + ladoMayor * altura + ladoMenor * altura)

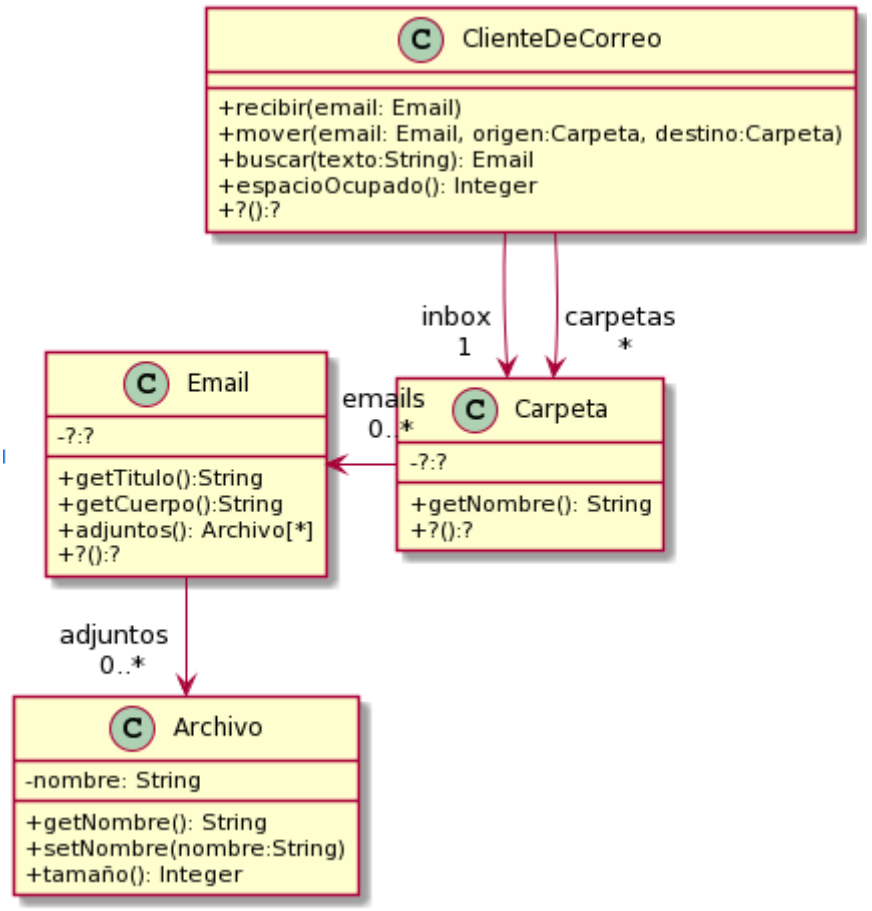
- Para obtener π, utilizamos Math.PI
- Para elevar un número a cualquier potencia, utilizamos Math.pow(numero: double, potencia: double). Ej: 8^2 = Math.pow(8, 2)

Observaciones adicionales

Probablemente note una similitud entre este ejercicio y el ejercicio de "Figuras y cuerpos" que hizo anteriormente. En ambos ejercicios usted podía construir cilindros y prismas rectangulares. Sin embargo las implementaciones varían. Discuta diferencias y similitudes con el ayudante.

Ejercicio 13. Cliente de correo con adjuntos

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 que viene como parámetro de la carpeta origen a la carpeta destino (asuma que el email está en la carpeta origen cuando se recibe este mensaje).
- En respuesta al mensaje #buscar retorna el primer email que encuentra 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, asuma que el tamaño de un archivo es el largo de su nombre.

Tareas

1. Complete el diseño y el diagrama de clases UML.
2. Implemente en Java de la funcionalidad requerida.

3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 14. Intervalo de tiempo

En Java, las fechas se representan normalmente con instancias de la clase java.time.LocalDate (https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html). Se pueden crear con varios métodos "`static`" como por ejemplo `LocalDate.now()`.

- Investigue cómo hacer para crear una fecha determinada, por ejemplo 15/09/1972.
- Investigue cómo hacer para determinar si la fecha de hoy se encuentra entre las fechas 15/12/1972 y 15/12/2032. Sugerencia: vea los métodos permiten comparar LocalDates y que retornan `booleans`.
- Investigue cómo hacer para calcular el número de días entre dos fechas. Lo mismo para el número de meses y de años Sugerencia: vea el método `until`.
  Tenga 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`

Tarea 1
Implemente 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:

```
public LocalDate getFrom()
“Retorna la fecha de inicio del rango”

public LocalDate getTo()
“Retorna la fecha de fin del rango”

public int sizeInDays()
“retorna la cantidad de di´as entre la fecha 'from' y la fecha 'to'”

public boolean includesDate(LocalDate other)
“recibe un objeto LocalDate y retorna true si la fecha esta´ entre el from y el to
del receptor y false en caso contrario”.
```

Tarea 2

1. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
2. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Tarea 3
Asumiendo que implementó la clase DateLapse con dos variables de instancia “from” y “to”, realice otra implementación de la clase para que su representación sea a través de los atributos “from” y “sizeInDays” y coloquela en otro paquete. Es decir, debe basar su nueva

implementación en estas variables de instancia solamente. Intente definir una interfaz java para que ambas soluciones la implementen.

Los cambios en la estructura interna de un objeto sólo deben afectar a la implementación de sus métodos. Estos cambios deben ser transparentes para quien le envía mensajes, no debe notar ningún cambio y seguir usándolo de la misma forma. Por lo tanto, los tests que implementó en la tarea 2 deberían pasar sin problemas.

Ejercicio 15. Alquiler de propiedades

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial

Necesitamos que usted implemente OOBnB, un sistema para publicar propiedades en alquiler, y para alquilarlas. Identifique objetos y responsabilidades. El sistema ofrece la siguiente funcionalidad:

Registrar usuarios: Se provee nombre, dirección, dni. El sistema da de alta el usuario. El sistema retorna el Usuario. El usuario no tiene propiedades en alquiler. El usuario no tiene ninguna reserva de propiedad. El usuario no ha alquilado nunca una propiedad.

Registrar una propiedad en alquiler: Se provee nombre, descripción, precio por noche, y dirección. Se provee el usuario propietario. El sistema da de alta la propiedad y la retorna. La propiedad no tiene ninguna fecha ocupada.

Buscar propiedades disponibles en un período: Se indica el período (fecha de inicio y fecha de fin). Retorna todas las propiedades que se encuentran disponibles desde la fecha de inicio (inclusive) hasta el día de fin (inclusive).

Hacer una reserva: Se indica la propiedad, el período y el usuario para quien se hace la reserva (el inquilino). Si la propiedad está libre, se genera la reserva (que queda registrada en el sistema). La propiedad pasa a estar ocupada en esas fechas. Si la propiedad no está libre no hace nada y retorna null. Ver notas al final de este ejercicio sobre cómo podría resolver este punto.

Calcular el precio de una reserva: dada una reserva, obtener el precio a partir del precio por noche de la propiedad y la cantidad de noches de la reserva.

Eliminar reserva: Dada una reserva, si la fecha de inicio de la reserva es posterior a la fecha actual se elimina la reserva. La propiedad pasa a estar disponible en esas fechas.

Obtener las reservas de un usuario: dado un usuario, obtener todas las reservas que ha efectuado (pasadas o futuras).

Calcular los ingresos de un propietario: dado un usuario, y dos fechas, obtener el monto total que conseguirá por todas las reservas, de todas sus propiedades, entre las fechas indicadas.

Notas sobre el diseño e implementación:
Para el manejo de los períodos de reserva puede considerar usar la implementación de DateLapse (**ejercicio 14 Intervalo de tiempo**). La clase DateLapse podría ser mejorada agregando un nuevo método:

```
/**
      Retorna true si el período de tiempo del receptor se superpone con el
      recibido por parámetro
**/
public boolean overlaps (anotherDateLapse: DateLapse)
```

Tareas

1. Complete el diseño y el diagrama de clases UML.
2. Implemente en Java de la funcionalidad requerida.
3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 16. Políticas de cancelación

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial

En el sistema de alquiler de propiedades del ejercicio 15 (OOBnB) se quiere introducir funcionalidad para calcular el monto que será reembolsado (devuelto) si se cancela una reserva. Eso cambia la la funcionalidad indicada anteriormente de la siguiente manera:

Registrar una propiedad en alquiler: Se provee nombre, descripción, precio por noche, y dirección. *Adicionalmente se indica la política de cancelación.* El sistema da de alta la propiedad y la retorna. La propiedad no tiene ninguna fecha ocupada.
La política de cancelación puede ser una de tres: flexible, moderada, o estricta.

Calcular el monto a reembolsar si se hiciera una cancelación: Dada una reserva y una fecha tentativa de cancelación, devuelve el monto que sería reembolsado. El cálculo se hace de la siguiente manera.

a) Si la propiedad tiene política de cancelación flexible, se reembolsará el monto total sin importar la fecha de cancelación (que de todas maneras debe ser anterior a la fecha de inicio de la reserva).
b) Si una propiedad tiene política de cancelación moderada, se reembolsará el monto total si la cancelación se hace hasta una semana antes y 50% si se hace hasta 2 días antes.
c) Si una propiedad tiene política de cancelación estricta, no se reembolsará nada (0, cero) sin importar la fecha tentativa de cancelación.

Actualice su diseño, implementación y tests.

Ejercicio 17. Facturación de llamadas

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial.

Se desea implementar un sistema de registro y facturación de llamadas telefónicas. El sistema ofrece la siguiente funcionalidad:

Agregar un número a la lista de números disponibles. Se provee un número de teléfono. El sistema lo agrega a la lista de números telefónicos disponibles. Asuma que el número de teléfono que se provee es nuevo (nunca fue utilizado).

Dar de alta como cliente a una persona física (un individuo). Se provee nombre, dirección, DNI. El sistema da de alta el cliente y le asigna un número telefónico de la lista de números de teléfonos disponibles. El número asignado deja de estar disponible. El sistema retorna el cliente.

Dar de alta como cliente a una persona jurídica (empresa, organismo, asociación, etc,). Se provee nombre, dirección, CUIT y tipo de persona jurídica (por ejemplo Sociedad Anónima, Repartición Provincial, etc.). El sistema da de alta el cliente y le asigna un número telefónico de la lista de números de teléfonos disponibles. El número asignado deja de estar disponible. El sistema retorna el cliente.

Registrar una llamada local. Se provee la fecha y hora de comienzo, la duración en minutos, el número del teléfono que llama y el del teléfono que recibe. El sistema guarda el registro de la llamada. El sistema retorna el registro de la llamada.

Registrar una llamada interurbana. Se provee la fecha y hora de comienzo, la duración en minutos, el número del teléfono que llama y el del teléfono que recibe. Se provee la distancia en kilómetros entre el que llama y el que recibe. El sistema guarda el registro de la llamada. El sistema retorna el registro de la llamada.

Registrar una llamada internacional. Se provee la fecha y hora de comienzo, la duración en minutos, el número del teléfono que llama y el del teléfono que recibe. Se provee el país de origen y país destino de la llamada. El sistema guarda el registro de la llamada. El sistema retorna el registro de la llamada.

Facturar las llamadas de un cliente. Se indica el cliente para el cual se quiere facturar. Se indican las fechas de inicio y fin del período a considerar. El sistema retorna una factura en la que consta: el cliente al que pertenece, la fecha de facturación, las fechas de inicio y fin del período, y el monto total de todas las llamadas que el cliente hizo, y que iniciaron en ese período.

Para el cálculo del costo de una llamada tenga en cuenta lo siguiente:

1. Las llamadas locales tienen un costo por minuto de duración (utilice $1).
2. Las llamadas interurbanas tienen un costo de conexión fijo (utilice $5), y un costo por minuto de duración que depende de la distancia (utilice $2 para menos de 100 km, $2.5 para distancias entre 100 km y 500 km, y $3 para distancias de más de 500 km).

3. Las llamadas internacionales tienen un costo por minuto que depende del país destino y de la hora de comienzo (el precio diurno de 8:00 a 20:00 es un valor, y el precio nocturno de 20:00 a 8:00 es otro). Por ahora utilice $4 como precio diurno para todos los países y $3 como precio nocturno para todos los países.
4. Las llamadas efectuadas por personas físicas tienen un 10% de descuento.

1) Diseñe (documente en un diagrama de clases UML) e implemente en Java toda la funcionalidad antes descripta.

2 - bonus) Es probable que los montos utilizados para los cálculos le hayan quedado fijos dentro del código (hardcoded). Piense qué pasaría si al facturar se proveyera (como un parámetro más) el "cuadro tarifario". ¿Cómo sería ese objeto? ¿Qué responsabilidad le podría delegar? ¿Cómo haríamos para tener montos diferentes para los distintos países en las llamadas internacionales?

3 - Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.

4 - Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 18. Liquidación de haberes

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial

Nos metemos en el negocio de los sistemas de gestión de empresas y, para ello, vamos a comenzar por desarrollar un módulo de liquidación de haberes. Debe ofrecer la siguiente funcionalidad:

Dar de alta un empleado: Se indica el nombre, apellido, CUIL y fecha de nacimiento. Se indica si tiene cónyuge a cargo. Se indica si tiene hijos a cargo. El sistema agrega el empleado a la nómina de la empresa. Se registra la fecha actual como fecha de inicio de la relación laboral del empleado.

Buscar un empleado: Se indica el CUIL del empleado. El sistema retorna al empleado con ese CUIL o null si no existe.

Dar de baja un empleado: Se indica el empleado a dar de baja. El sistema lo quita de la nómina de la empresa.

Cargar el contrato de un empleado: Se indica el empleado, la fecha de inicio del contrato, la fecha de fin (si corresponde) y algunos valores adicionales dependiendo del tipo de contrato. Hay dos tipos de contratos:

1. Si el contrato es "por horas", se indica el valor-hora acordado, y el número de horas que trabajará por mes. También se indica la fecha de fin del contrato.
2. Si el contrato es "de planta", se indica el sueldo mensual acordado, el monto acordado por tener cónyuge a cargo, y el monto acordado por tener hijos a cargo. Estos contratos no tienen fecha de fin (nunca se vencen).

El sistema registra el contrato creado para el empleado. Pueden existir múltiples contratos creados para un mismo empleado, sin embargo un empleado solo puede tener un único contrato activo (no vencido) a la vez. El contrato activo para el caso de contrato “de planta” es el único contrato vigente. Para un contrato “por horas”, se considera activo aquel cuya fecha de fin sea posterior a la fecha actual.

Obtener empleados con contratos vencidos. El sistema devuelve la lista de todos aquellos empleados cuyo contrato actual se encuentre vencido. Si para un empleado existiese más de un contrato, el contrato con fecha de inicio más reciente es el considerado actual, dicho contrato puede estar vigente (si no tiene fecha de fin o si la fecha de fin es posterior a la fecha actual), o vencido (para los que tienen fecha de fin, cuando dicha fecha es inferior o igual a la fecha actual)

Generar recibos de sueldo. Por cada empleado (con contrato activo, es decir sin vencer) el sistema genera un recibo de sueldo. El sistema devuelve los recibos de sueldo. De un recibo de sueldo puede obtenerse la siguiente información: el nombre, apellido, CUIL y antigüedad en la empresa del empleado al que pertenece el recibo; la fecha en la que fue generado el recibo; y el monto total que le corresponde cobrar al empleado.
El monto se calcula en dos pasos, el básico y la antigüedad. El básico se calcula de la siguiente forma:

1. Si su contrato es por horas fijas, el monto a cobrar es el valor-hora acordado multiplicado por el número de horas que trabaja por mes.
2. Si su contrato es de planta, el monto a cobrar es el sueldo mensual acordado, más el monto acordado por tener cónyuge a cargo (si es que tiene cónyuge a cargo), más el monto acordado por tener hijos a cargo (si es que tiene hijos a cargo).

La antigüedad se calcula como un porcentaje del básico. Aumenta automáticamente cuando se alcanza cierta antigüedad, en función de esta escala: 5 años 30%, 10 años 50%, 15 años 70%, 20 años 100%. Tenga en cuenta que la antigüedad de un empleado es la suma de las duraciones de los contratos.

Su tarea es diseñar y programar en Java lo que sea necesario para ofrecer la funcionalidad antes descrita. Se espera que entregue los siguientes productos.

1. Diagrama de clases UML.
2. Implementación en Java de la funcionalidad requerida.
3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 19. Mercado de Objetos

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial

Queremos programar en objetos una versión simplificada de un mercado on-line similar a e-Bay o MercadoLibre.

El sistema ofrece la siguiente funcionalidad (que usted deberá implementar)

- **Registrar un vendedor:** Se indica el nombre del vendedor y su dirección. Se agrega el vendedor y se lo retorna.
- **Buscar un vendedor:** Se indica el nombre del vendedor que se desea buscar/recuperar. Si existe lo retorna. Si no, retorna null. Asuma que no hay nombres repetidos.
- **Registrar un cliente:** Se indica el nombre del cliente y su dirección. Se agrega cliente y se lo retorna. Asuma que no hay nombres repetidos.
- **Buscar un cliente:** Se indica el nombre del cliente que se desea buscar/recuperar. Si existe lo retorna. Si no, retorna null.
- **Poner un producto a la venta:** Se indica el nombre del producto, su descripción, su precio, la cantidad de unidades disponibles y el vendedor. Retorna el producto
- **Buscar un producto:** Se indica el nombre del producto que se desea buscar/recuperar. Retorna una colección con los productos que tienen ese nombre o una colección vacía.
- **Crear un pedido.** Se indica el cliente. Se indica el producto y la cantidad solicitada. Se indica la forma de pago elegida y el mecanismo de envío elegido. Si hay suficientes unidades disponibles del producto, el sistema registra el pedido y actualiza la cantidad de unidades disponibles del producto. Si no hay suficientes unidades disponibles, no se hace nada.
    - Las *opciones de pago posibles* son: "al contado" o "6 cuotas". A futuro podrían agregarse otras formas de pago.
    - Los *mecanismos de envío posibles* son: "retirar en el comercio", "retirar en sucursal del correo", ó "exprés a domicilio". A futuro podrían agregarse otros mecanismos de envío.
- **Calcular el costo total de un pedido.** Dado un pedido, se retorna su costo total que se calcula de la siguiente forma: (precio final en base a la forma de pago seleccionada) + (costo de envío en base al mecanismo de envío seleccionado).
    - si la forma de pago es "al contado", el precio final es el que se indica en el producto
    - si la forma de pago es "6 cuotas", el precio final se incrementa en un 20%
    - si el mecanismo de envío es "retirar en el comercio" no hay costo adicional de envío.
    - si el mecanismo de envío es "retirar en sucursal del correo" el costo es $50.
    - si el mecanismo de envío es "express a domicilio" el costo es $0.5 por Km de distancia entre la dirección del vendedor y del cliente. Asuma que existe una clase Mapa, cuyas instancias entienden el mensaje #distanciaEntre que recibe dos direcciones y retorna la distancia en Km entre ellas. Por ahora trabaje con una implementación suya (de pruebas) de esa clase que siempre retorna 100 (sin importar las direcciones).

Su tarea es diseñar y programar en Java lo que sea necesario para ofrecer la funcionalidad antes descrita.

1. Diagrama de clases UML.
2. Implementación en Java de la funcionalidad requerida.
3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 20. Farolas con focos

Se desea extender el nivel de detalle de la red de alumbrado (Ejercicio 6) para que ahora se pueda incluir en las farolas información del foco (o lamparita) que posee cada una. De un foco se desea conocer la marca de la empresa que lo fabricó y la cantidad de veces máxima que puede encenderse. A esto último lo llaman ciclos de encendido. Cada vez que a un foco apagado se lo enciende se completa un ciclo de encendido. Por ejemplo, si a un foco se lo enciende, apaga, y enciende otra vez se le completan 2 ciclos de encendido. Cuando un foco supera la cantidad de ciclos de encendido con el que fue fabricado se dice que el foco se encuentra vencido.

Actualice el ejercicio de farolas con los siguientes casos de uso:

```
/*
* Crear una farola que posee un foco fabricado por fabricante con cantidad
de ciclos el valor de cantidadDeCiclos. Debe inicializarla como apagada
*/
public Farola (String fabricante, int cantidadDeCiclos)
/*
* Crea la relación de vecinos entre las farolas. La relación de vecinos
entre las farolas es recíproca, es decir el receptor del mensaje será vecino
de otraFarola, al igual que otraFarola también se convertirá en vecina del
receptor del mensaje
*/
public void pairWithNeighbor( Farola otraFarola )
/*
* Retorna sus farolas vecinas
*/
public List<Farola> getNeighbors ()

/*
* Si la farola no está encendida, la enciende, contabiliza el ciclo de
encendido y propaga la acción.
*/
public void turnOn()

/*
* Si la farola no está apagada, la apaga y propaga la acción.
*/
public void turnOff()

/*
* Retorna una lista con las farolas que están en la red de la farola
receptora y que poseen focos vencidos. Incluyendo el chequeo entre las
farolas vecinas y las vecinas de estas propagando en toda la red.
*/
public List<Farola> farolasConFocosVencidos()
```

Tareas

1. Complete el diseño y el diagrama de clases UML.
2. Implemente en Java de la funcionalidad requerida.
3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.

Ejercicio 21. PoolCar

Nota: este ejercicio es del estilo de los que encontrarán en la evaluación parcial

PoolCar es una aplicación que permite a sus usuarios compartir costos en viajes de larga distancia.

La aplicación conoce a los usuarios de los cuales se guarda su nombre y saldo. El saldo les permitirá pagar los viajes que realicen. Hay dos tipos de usuarios: conductores y pasajeros. El sistema tiene una restricción: un usuario se registra a la aplicación como conductor o como pasajero, y no puede cambiar su rol. Cada conductor tiene un vehículo y solamente puede viajar en él. De cada vehículo se conoce: dueño/conductor, descripción, capacidad de pasajeros (incluido el conductor), año de fabricación y el valor de mercado.

Cuando PoolCar registra un viaje, lo hace guardando: origen, destino, costo total, vehículo y fecha de viaje.

Los usuarios pasajeros pueden registrarse a los viajes siempre y cuando haya capacidad en el vehículo, se registren al menos dos días antes de la fecha del mismo y no tengan su saldo en rojo (menos de 0). Cuando un viaje es registrado PoolCar agrega (automáticamente) al dueño del vehículo como pasajero del viaje. El conductor compartirá los gastos del viaje con los demás pasajeros.

Todos los días PoolCar procesa los viajes que empiezan al día siguiente. Al procesar un viaje se debe descontar el saldo correspondiente a los integrantes (conductor y pasajeros) del mismo. El costo total del viaje se reparte en partes iguales entre todos los integrantes que participaron. Sin embargo, al momento de procesar un viaje, PoolCar realiza una bonificación que se calcula de manera distinta para los conductores que para los pasajeros. Del monto correspondiente al conductor, se le bonifica el 0.1% del valor del vehículo utilizado para el viaje. Del monto correspondiente a un pasajero, PoolCar bonifica $500 si el usuario hizo un viaje en el pasado. Es posible que, luego de procesar el viaje, el saldo de un usuario quede en rojo (saldo < 0).

<u>Su solución debe implementar, al menos, el siguiente protocolo:</u>

Dar de alta un usuario conductor: se provee el nombre y el vehículo de su propiedad. El sistema da de alta al conductor y lo retorna. El sistema registra al vehículo como perteneciente al conductor. El conductor tiene su saldo en 0.

Dar de alta un usuario pasajero: se provee el nombre. El sistema da de alta al pasajero y lo retorna. El pasajero tiene su saldo en 0.

Cargar saldo para un usuario: dado un usuario y un monto, incrementa el monto indicado en el saldo en la cuenta del usuario. PoolCar cobra una comisión (extra sobre el saldo). Para el caso de los conductores, la comisión es siempre del 1% si el auto tiene menos de 5 años. Si no, es del 10%. En el caso de un pasajero, se cobra una comisión del 10% solo si no realizó ningún viaje los últimos 30 días.

Dar de alta un viaje desde un origen a un destino: se provee el origen, el destino, el costo total, el vehículo con el que se realizará el viaje y la fecha. El sistema registra el viaje y lo retorna.

Listar los viajes que comienzan al día siguiente: retorna un listado de todos los viajes cuya fecha de viaje es el día siguiente.

Procesar los viajes: permite procesar todos los viajes que comienzan el día siguiente, siguiendo la especificación anterior.

Retornar los usuarios registrados: retorna un listado con todos los usuarios registrados (esto es, todos los conductores y pasajeros registrados en el sistema). El listado debe estar ordenado en forma descendente según el saldo de los usuarios.

Cargar un monto de regalo a los usuarios registrados: se indica el monto a incrementar y el sistema incrementa el saldo indicado a todos los usuarios registrados.

Registrar pasajero para un viaje: dado un pasajero y un viaje, registra ese pasajero para que pueda compartir ese viaje, con las reglas explicadas anteriormente.

Actividades:

Su tarea es diseñar y programar en Java lo que sea necesario para ofrecer la funcionalidad antes descrita. Se espera que entregue los siguientes productos.

1. Diagrama de clases UML.
2. Implementación completa (ver notas) en Java de la funcionalidad requerida.
3. Diseñe los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría para la funcionalidad que considere necesaria.
4. Implemente utilizando JUnit los tests automatizados diseñados en el punto anterior.