Cuadernillo Semestral de Actividades

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.

Fecha de la última edición: 1 de Octubre de 2018.
(Es probable que sufra algunos cambios menores o se agreguen ejercicios por eso deben estar atentos a la fecha de la última edición)

Ejercicio 1: WallPost

Primera parte

Se está construyendo una red social como Facebook o twitter. Definimos 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 Wallpost es subclase de Object.

Cree un nuevo paquete *Objetos1-Wallpost-Model*

Dentro de ese paquete defina la clase Wallpost en Smalltalk, para que entienda los siguientes mensajes:

#text

"Retorna el texto descriptivo de la publicación"

#text: descriptionText

"Setea el texto descriptivo de la publicación"

#likes

"Retorna la cantidad de “me gusta”"

#like

"Incrementa la cantidad de likes en uno"

#dislike

"Decrementa la cantidad de likes en uno. Si ya es 0, no hace nada"

#isFeatured

"Retorna true si el post está marcado como destacado, false en caso contrario"

#toggleFeatured

"Cambia el post del estado destacado a no destacado y viceversa"

#initialize

"Inicializa el estado de las variables de instancia del Wallpost. Luego de la invocación el Wallpost debe tener como texto: “Undefined post”, no debe estar marcado como destacado y la cantidad de “Me gusta” deben ser 0."

Segunda parte

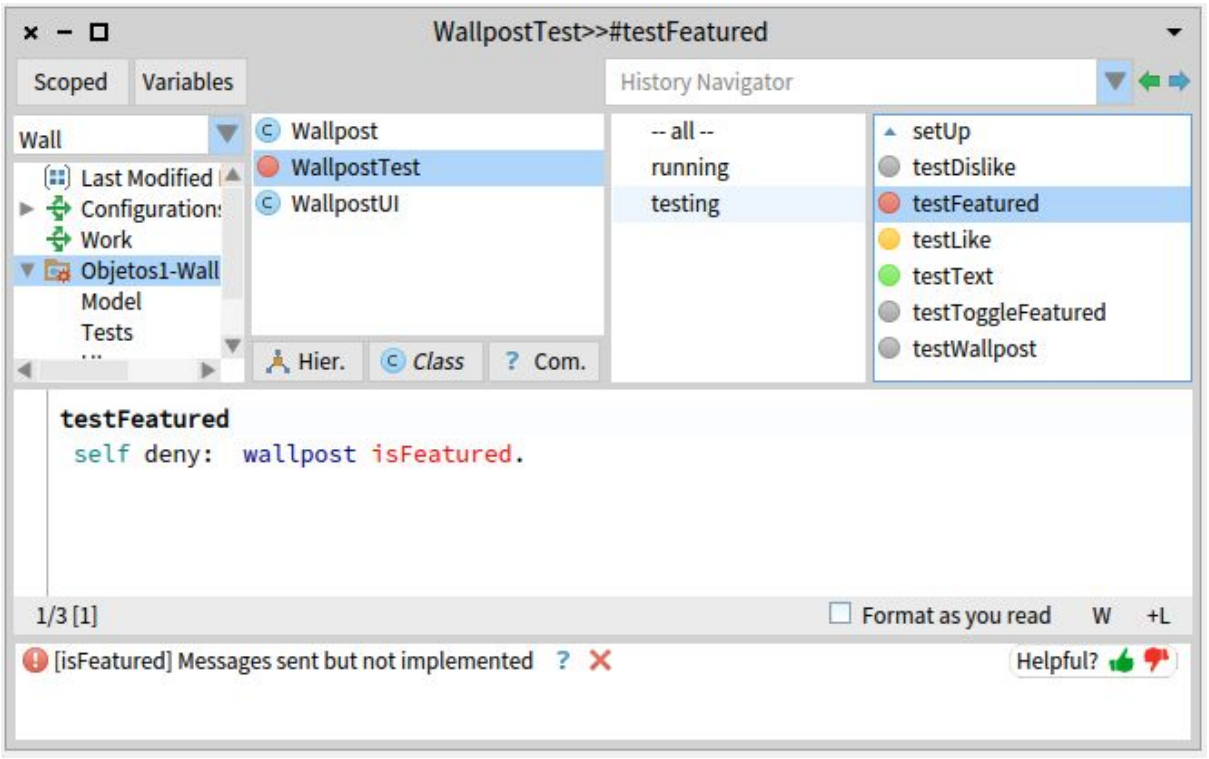
Evalúe la siguiente expresión en un Playground.

```
(IceRepositoryCreator new
    url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
    createRepository) updatePackage: 'Objetos1-WallpostSkeleton'.
```

Esa expresión creará un paquete Objetos1-WallpostSkeleton y descargará en el mismo las pruebas (tests) que utilizaremos para verificar el código que ha escrito.

Utilice los tests provistos por la cátedra para comprobar que su implementación de Wallpost es correcta. En la siguiente figura se observa el detalle de las clase de pruebas WallpostTest. Para ejecutar los tests simplemente haga click sobre la burbuja que se encuentra a la derecha del

nombre de la clase. Si la burbuja es de color gris, significa que el test no ha sido ejecutado todavía. Si es rojo significa que hubo errores. Si es amarillo significa que los resultados no son los esperados. Si es verde, su código ha pasado el test. Siéntase libre de investigar la implementación de la clase de test. Ya veremos en detalle cómo implementarlas.



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 y dislike, marcarlo como destacado).

Para abrir la ventana puede evaluar la siguiente expresión en el Playground:

```
WallpostUI on: (Wallpost new)
```

En la expresión que evaluó para abrir la ventana WallpostUI on: (Wallpost new) se instancian 2 objetos, el wallpost y la ventana. Discuta con un ayudante:

- ¿En qué difieren las instanciaciones?

En la primer parte de este ejercicio ud. implementó el método #initialize, pero, ¿quien lo invoca?

Ayuda: coloque un breakpoint en el método initialize para ver quién lo invoca.
Ayuda de la ayuda: para poner un breakpoint agregue la sentencia self halt. al código del método #initialize.

Ejercicio 2: Balanza Electrónica

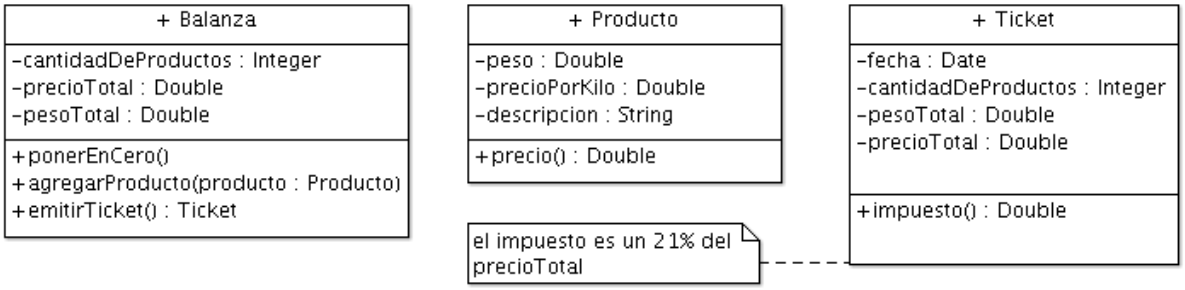
En taller de programación programó (en Java) una balanza electrónica. Volveremos a programarla, ahora en Smalltalk, con algún requerimiento adicional.

En términos generales, la Balanza electrónica recibe productos (uno a uno), y calcula dos totales: peso total y precio total. La balanza no guarda los productos. Luego emite un ticket que indica número de productos considerados, peso total, precio total.

Implemente:

Cree un nuevo paquete Objetos1-BalanzaElectronica.

En ese paquete 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).

Al programarlas en Pharo, todas son subclases de Object.

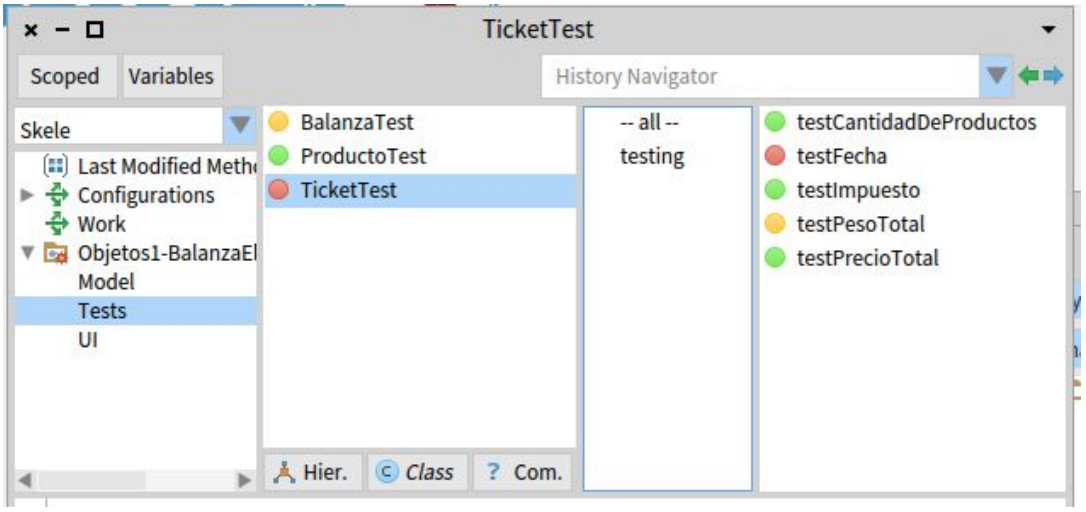
Probando su implementación:

Evalúe la siguiente expresión en un Playground.

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-BalanzaElectronicaSkeleton'.
```

Esa expresión creará un paquete Objetos1-BalanzaElectronicaSkeleton y descargará en el mismo las pruebas (tests) que su código deberá pasar.

Si todo salió bien, su implementación debería pasar las pruebas que definen las clases que estan el el tag "Tests" del paquete que instaló al principio de este ejercicio. El propósito de estas clases es ejercitar una instancia de la clase Balanza y verificar que se comporta correctamente.



En la figura se observa el detalle de las clases de prueba. Para ejecutar los tests simplemente haga click sobre la burbuja que se encuentra a la derecha del nombre de cada clase. Si la burbuja es de color gris, significa que el test no ha sido ejecutado todavía. Si es rojo significa que hubo errores. Si es amarillo significa que los resultados no son los esperados. Si es verde, su código ha pasado el test. . Siéntase libre de investigar la implementación de estas clases.

Interactuando directamente con los objetos

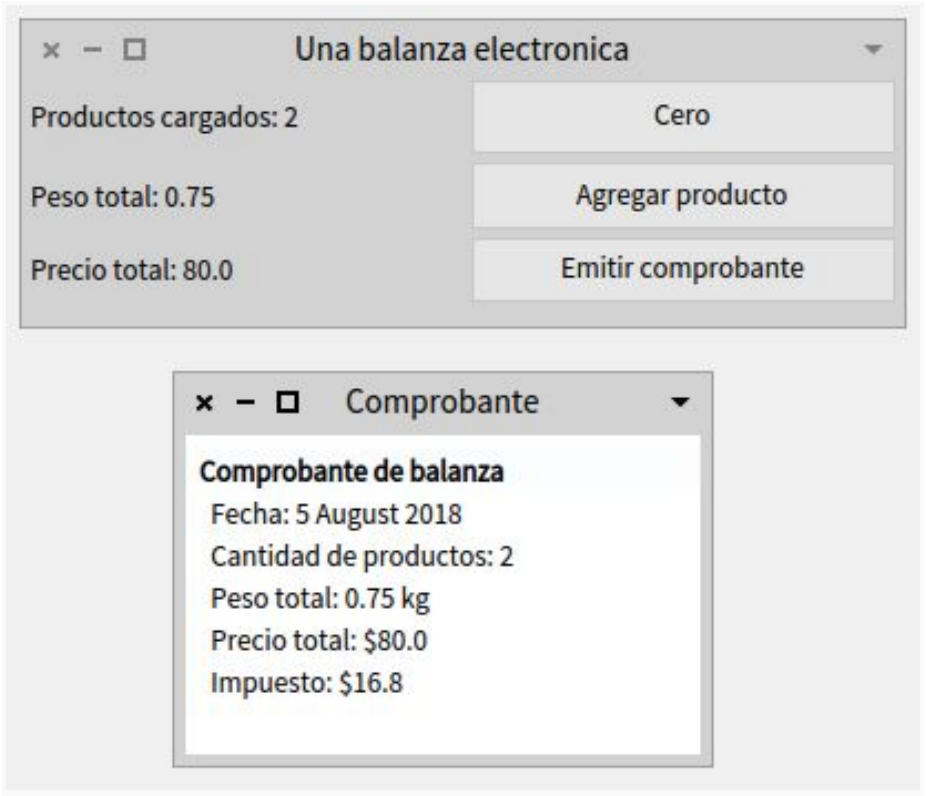
Las siguientes expresiones muestran cómo hacer pruebas en un Playground. Esto es útil cuando se quiere entender cómo funciona algún objeto o se quiere experimentar. Para probar que los objetos funcionan bien usamos los tests (que ya aprenderemos a escribir)

```
balanza := Balanza new.
producto := Producto new.
producto descripcion: 'Crema'.
producto precioPorKilo: 43.
producto peso: 0.25.
balanza agregarProducto: producto.
producto := Producto new.
producto descripcion: 'Queso'.
producto precioPorKilo: 10.
producto peso: 2.
balanza agregarProducto: producto.
balanza emitirTicket.
```

La interfaz gráfica de la balanza

Para darle una idea de como una interfaz gráfica trabajaría con esos objetos, se ofrece una implementación ejemplo (que se descargó junto con los tests. Si su implementación pasó los tests, puede utilizar la siguiente expresión para comenzar a la interfaz.

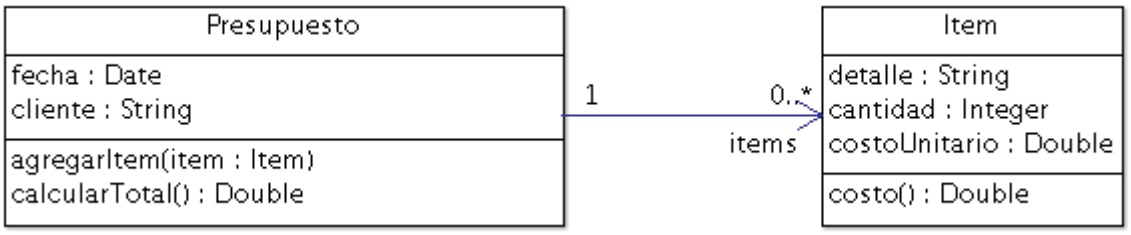
```
interfaz := BalanzaComposableModel new.
interfaz balanza: Balanza new.
interfaz openWithSpec
```


Una balanza electronica
Productos cargados: 2
Peso total: 0.75
Precio total: 80.0
Cero
Agregar producto
Emitir comprobante
Comprobante
Comprobante de balanza
Fecha: 5 August 2018
Cantidad de productos: 2
Peso total: 0.75 kg
Precio total: $80.0
Impuesto: $16.8

Ejercicio 3: Presupuestos

Defina un paquete Objetos1-Presupuesto-Model 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 instancias de cada clase?
- ¿Qué variables inicializa y cómo?



Probando su código:
Evalúe la siguiente expresión en un Playground.

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-PresupuestoSkeleton'.
```

Encontrará un paquete Objetos1-PresupuestoSkeleton. En el mismo existe un tag "Tests" en el que hay dos clases que implementan los tests que su implementación deberá pasar.

Utilice los tests provistos para confirmar que su implementación ofrece la funcionalidad esperada. Siéntase libre de explorar las clases de test para intentar entender qué es lo que hacen.

En un playground instancie un presupuesto y agregue dos items al mismo. Luego calcule el costo total del presupuesto.

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 #productos, que retorne 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 #productos .

¿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?

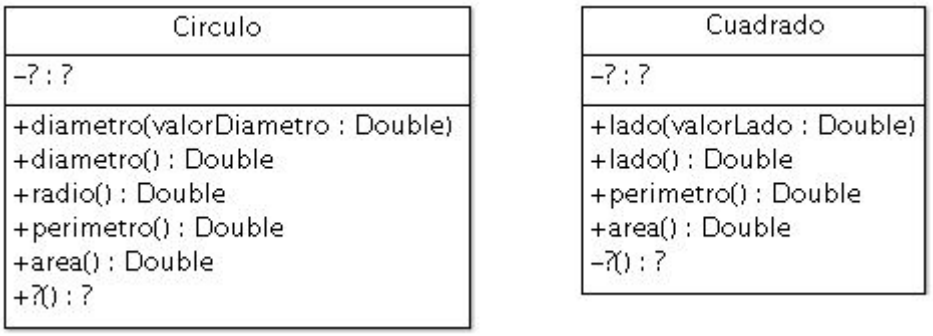
Ejercicio 4: Figuras y cuerpos

Figuras en 2D

Defina un nuevo paquete Objetos1-FigurasYCuerpos-Model

En taller de programación definió clases para representar figuras geométricas. Retomaremos ese ejercicio para implementarlo en Smalltalk (ahora vamos a 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.

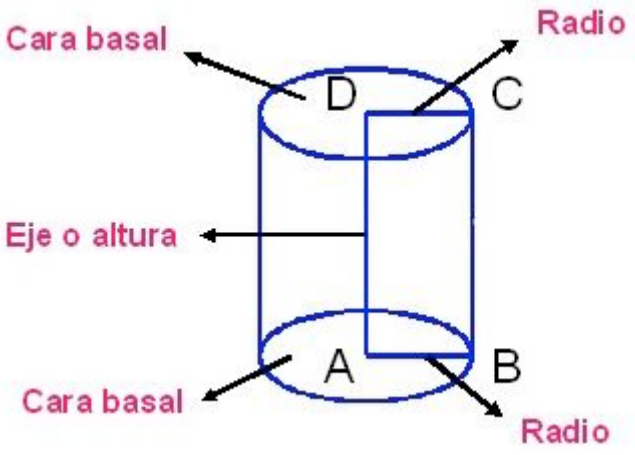


Fórmulas y mensajes útiles:

- 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)
- Para elevar un numero al cuadrado, le enviamos el mensaje #squared (8 squared)

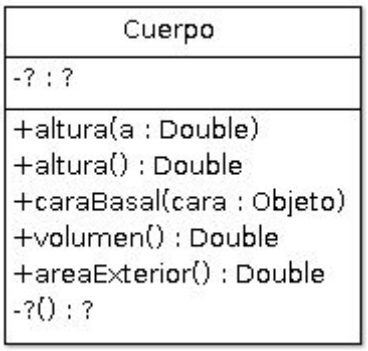
Cuerpos en 3D

Ahora que tenemos Círculos y Cuadrados, podemos usarlos para construir cuerpos (en 3D) y calcular su volumen y área exterior. Vamos a pensar a un cilindro como "un cuerpo que tiene un círculo como su cara basal y que tiene una altura (vea la siguiente imágen)" .



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 cuerpo. Decida usted qué variables de instancia son necesarias. Cuerpo es subclase de Object.



Fórmulas útiles:

- El área exterior de un cuerpo es:
  2* area-cara-basal + perimetro-cara-basal * altura-del-cuerpo
- El volumen de un cuerpo es: area-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 mas al respecto.

Pruebas automatizadas

Evalúe la siguiente expresión en un Playground.

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-FigurasYCuerposSkeleton'.
```

Esa expresión creará un paquete Objetos1-FigurasYCuerposSkeleton y descargará en el mismo las pruebas (tests) que su código deberá pasar.

Siguiendo los ejemplos de ejercicios anteriores, ejecute las pruebas automatizadas provistas. 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?

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

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:

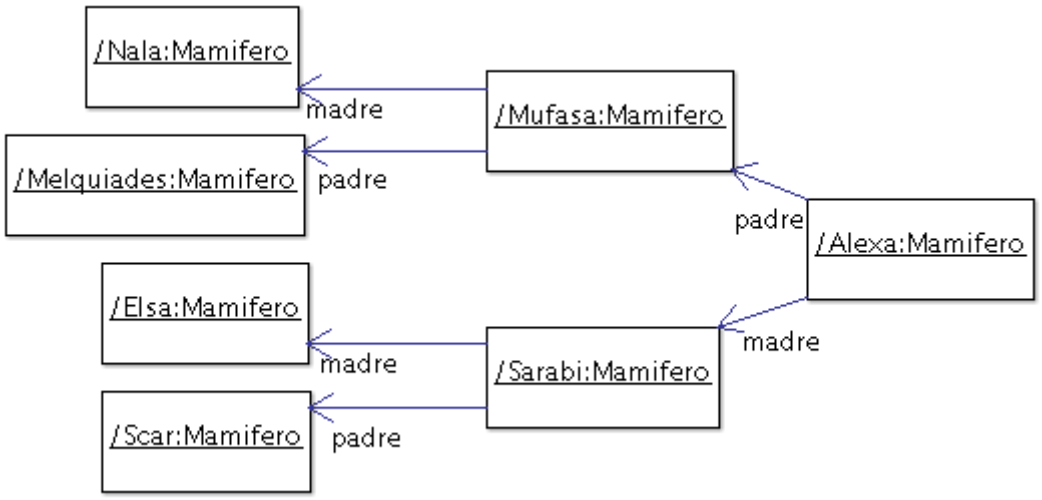
a) Modelar en objetos y programar en Pharo la clase Mamífero (como subclase de Object). El siguiente diagrama de clases (incompleto) nos da una idea de los mensaje que un mamífero entiende. *Deje #tieneAncestro para el final y discuta su solución con el ayudante.*

b) En un playground, instancie los objetos que se aprecian en el siguiente diagrama. Todos son de la misma especie, "Panthera leo".

Utilice el inspector para confirmar que lo que instanció es lo que se pide.

En el diagrama se puede apreciar el nombre/identificador de cada uno de ellos (por ejemplo Nala, Mufasa, Alexa, etc).

Para crear una fecha, envíe a la clase Date el mensaje #newDay:month:year: - Solo preste atención a que las fechas de nacimiento sean anteriores a las de sus crías.



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

d) Evalúe la siguiente expresión para instalar casos de prueba que hemos preparado para usted. Luego ejecute los tests como lo hizo en ejercicios anteriores. Si alguno falla, consulte con el ayudante.

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-GenealogiaAnimalSkeleton'.
```

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 Pharo, la clase Farola, como subclase de Object, con los siguientes métodos:

`#initialize`
`"Inicializa a la farola como apagada"`

`#pairWithNeighbor: otraFarola`
`"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."`

`#turnOn`
`"Si la farola no está encendida, la enciende y propaga la acción."`

`#turnOff`
`"Si la farola no está apagada, la apaga y propaga la acción."`

`#isOn`
`"Retorna true si la farola está encendida."`

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

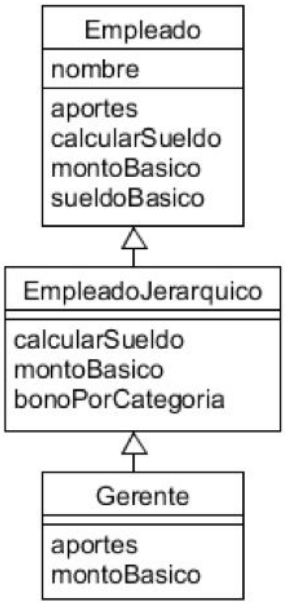
Para instalar los tests, evalúe la siguiente expresión:

```
(IceRepositoryCreator new
    url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
    createRepository) updatePackage: 'Objetos1-FarolasSkeleton'.
```

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

Ejercicio 7: Method lookup con empleados

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



Empleado	EmpleadoJerarquico	Gerente
sueldoBasico ^self montoBasico - self aportes	**calcularSueldo** ^super sueldoBasico + self bonoPorCategoria	**aportes** ^self montoBasico * 0.05
montoBasico ^ 700	**montoBasico** ^ 850	**montoBasico** ^1000
aportes ^10	**bonoPorCategoria** ^ 200	

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

```
| gerente |
gerente := Gerente new.
gerente aportes
```

```
| gerente |
gerente := Gerente new.
gerente calcularSueldo
```

a) Liste los métodos que son ejecutados como resultado del envío del último mensaje (por ejemplo, método #aportes de la clase X, ...)
b) Responda qué retorna la última expresión en cada caso

Ejercicio 8: Subclase de TestCase para probar la clase Set (Conjunto)

Implemente el ejemplo provisto en la sección 6.1 (Writing a test in 2 minutes) del libro "Learning Object-Oriented Programming, Design and TDD with Pharo". En el ejemplo se escribe un test de unidad (incompleto) para la clase Set (conjunto).

En el ejemplo solo se chequea que:

1) si agrego un elemento a un conjunto vacío, pasa a tener un solo elemento
2) si vuelvo a agregar el mismo elemento, el tamaño del conjunto no cambia

Extienda el ejemplo con un test llamado #testConstructor para que compruebe que al crear un conjunto, el mismo responde al mensaje #isEmpty con true.

Ahora responda:

1) Para implementar los tests tuvo que definir una nueva clase: ¿De qué clase es subclase?
2) Para hacer comprobaciones tuvo que enviar a "self" los mensajes #assert:equals: , #assert: y #deny: ¿Donde están implementados los métodos que corresponden a esos mensajes? ¿Por qué los entiende su objeto si usted nunca los implementó?
3) En los métodos que hacen las pruebas,
    a) ¿Quien es "self"?
    b) ¿De qué clase es instancia?
    c) En respuesta al mensaje #assert:equals: su objeto de test ejecuta un método que hereda, ¿cierto?
    d) Busque la implementación del método #assert:equals: que su objeto hereda. ¿Donde está implementado?
    e) El método #assert:equals: método hace referencia a "self". Cuando su objeto ejecuta el método #assert:equals: que hereda, ¿quién es self? ¿de qué clase es?

Ejercicio 9: FilteredSet

Smalltalk tiene la clase Set. Un Set solo agrega un objeto si el mismo no existe ya en la colección. Podemos imaginar que el método #add: en Set chequea antes de agregar el elemento.

Los bloques smalltalk (instancias de BlockClosure) puede utilizarse como condiciones. Por ejemplo, el bloque [:elem | elem > 1] retorna true si se evalúa enviando como parámetro un objeto mayor a 1. Para evaluar un bloque que espera un parámetro le enviamos el mensaje #value: . Vea el siguiente ejemplo.

```
condition := [:elem | elem > 20].
condition value: 10 "si hacemos print-it de esto, obtenemos false".
condition value: 30 "si hacemos print-it de esto, obtenemos true".
condition value  "si hacemos print-it de esto, se observará un error porque faltó un parámetro".
```

Implemente una nueva clase FilteredSet. Los FilteredSet se comportan casi exactamente igual a los Set. Sin embargo tienen un bloque (llamado condition) que utilizan para determinar si el elemento que se quiere agregar será agregado o no. El siguiente ejemplo muestra cómo crear, configurar y utilizar un FilteredSet que solo acepta strings de tamaño mayor a 20.

```
sampleSet := FilteredSet condition: [:elem | elem size > 20].
sampleSet add: 'Este no entra'.
sampleSet add: 'Este si entra porque tiene más de veinte caracteres. Es realmente muy largo'.
```

Ejercicio 9-bis: 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).

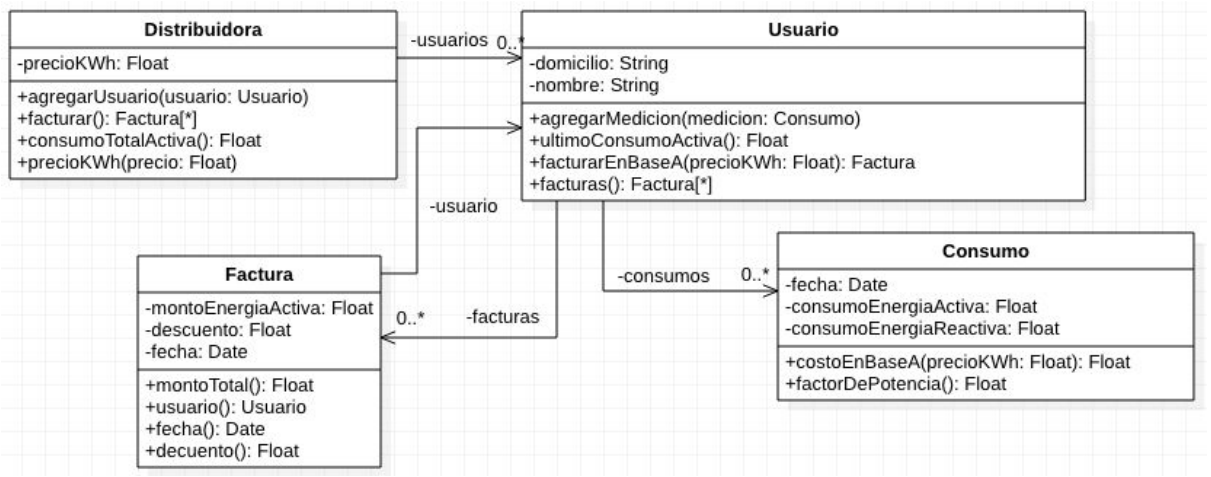
Para emitir la factura de un cliente se tiene en cuenta su último consumo y se calcula su factor de potencia para determinar si hay alguna bonificación para aplicar. El costo del consumo se calcula multiplicando el consumo de energía activa por el precio del kwh (Kilowatt/hora) de la empresa. La energía reactiva no tiene costo para el usuario. Si el factor de potencia 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:

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

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

El siguiente diagrama de clases muestra el diseño para este problema.



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 un playground (secuencia de expresiones) muestre como hacer para:

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

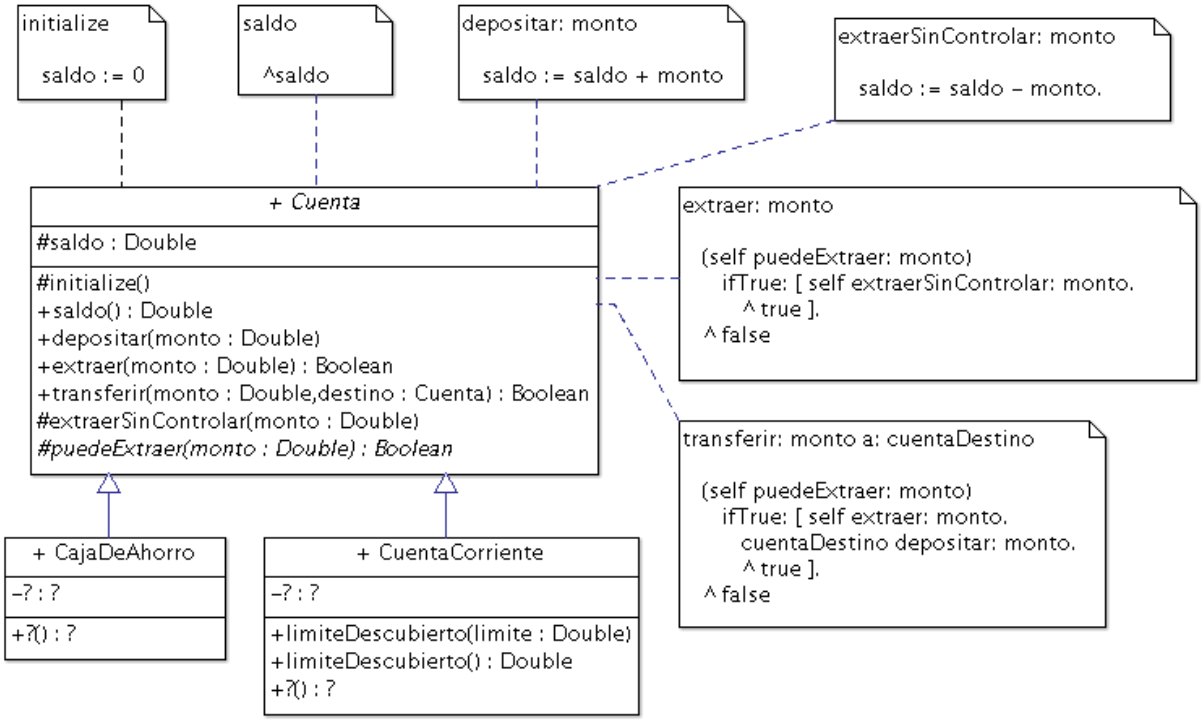
Solución de referencia

Evalúe la siguiente expresión para descargar una solución de referencia para este ejercicio:

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-DistribuidoraSolution'.
```

Ejercicio 10: 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 (tengalo 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 es 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 implementar el método #initialize).

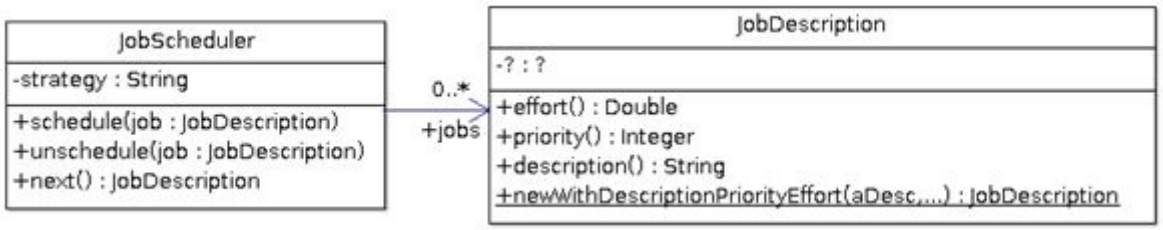
Tarea B: Escriba los tests de unidad que crea necesarios para garantizar que su implementación funciona adecuadamente.

Tarea C: 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 #tranferir:a: que se ven en el diagrama, ¿quien es self? ¿Puede decir que que clase es self?
c) ¿Por qué decidimos los métodos #initialize, #puedeExtraer: y #extraerSinControlar: tengan visibilidad "protegido"?
d) ¿Se puede transferir de una caja de ahorro a una cuenta corriente y viceversa? ¿por qué?
e) ¿Como implemento en Pharo un método abstracto? ¿Es obligatorio implementarlos? ¿Pharo fuerza a las subclases a implementar los métodos abstractos?

Ejercicio 11: 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 #schedulle: recibe un Job (trabajo) y lo **agrega al final** a la colección de trabajos pendientes.
- el mensaje #next determina cual 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.

1. Secuencia de ifs (o sentencia switch o case) para implementar alternativas de implementación de un mismo comportamiento
2. Código duplicado

Evalúe la siguiente expresión para obtener el código que corresponde a la implementación actual (y sus tests).

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Objetos1-JobSchedulerToRefactor'.
```

Utilice lo aprendido (en particular en relación a herencia y polimorfismo) para eliminar esos problemas. Siéntase libre de agregar nuevas clases como considere necesario. También puede cambiar la forma en la que los objetos se crean e inicializan.

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

Ejercicio 12: 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 y 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:

  'nombreDeLaAccion cantidadDeAcciones valorActualPorAccion'
- La clase PlazoFijo tiene las siguientes variables de instancia:

  'fechaDeConstitucion montoDepositado porcentajeDeInteresDiario'
- La clase Inversor tiene las siguientes variables de instancia:

  'nombre inversiones'

La variable inversiones de la clase Inversor es una colección con instancias de cualquiera de los dos clases de inversiones, y pueden estar mezcladas.

Cuando se quiere saber cuánto dinero representan las inversiones del inversor, se envía al mismo el mensaje #valorActual

1) Implemente en Smalltalk 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 (en la que se hace el cálculo)
- el valor actual de una InversiónEnAcciones se calcula multiplicando el número de acciones por el valor actual de las mismas.

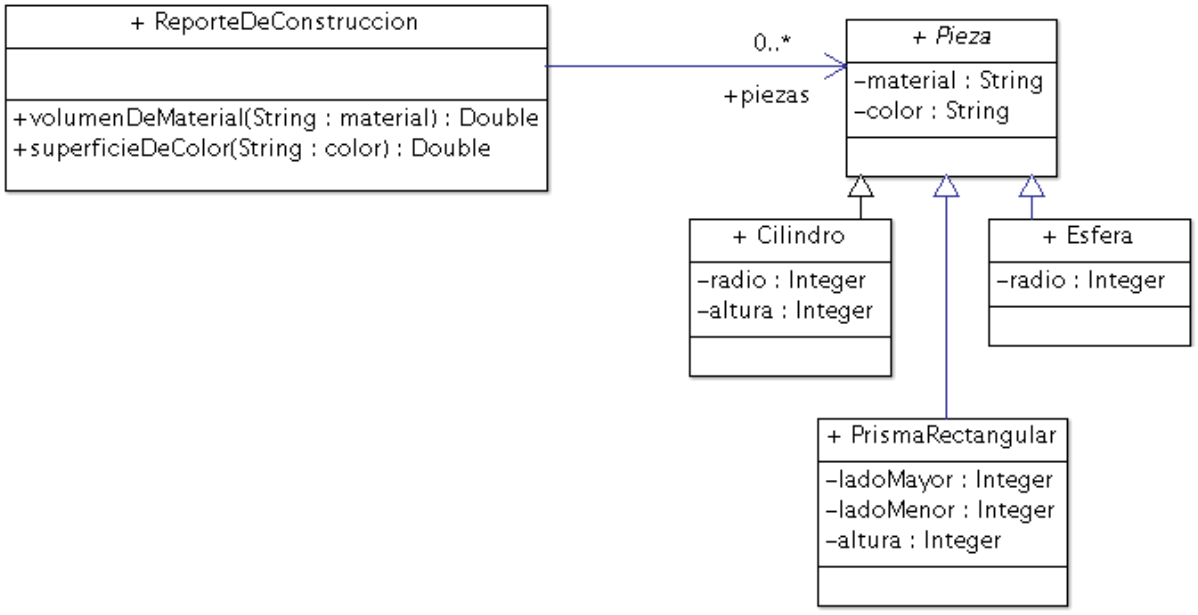
Recordatorio: No olvide la inicialización.

2) Construya un diagrama UML de clases para documentar su solución.

Ejercicio 13: 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 para determinar la cantidad de pintura que utilizó para pintar las piezas.

El siguiente diagrama de 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 que se exponen a continuación:

volumenDeMaterial: nombreDeUnMaterial
"Recibe como parámetro el un material (un string, por ejemplo 'Hierro'). Retorna la suma de los volúmenes de todas las piezas hechas en ese material"

superficieDeColor: unNombreDeColor
"Recibe como parametro 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: $\pi * radio^2 * h$.
Superficie de un cilindro: $2 * \pi * radio * h + 2 * \pi * radio^2$
Volumen de una esfera: $\frac{4}{3} * \pi * radio^3$.

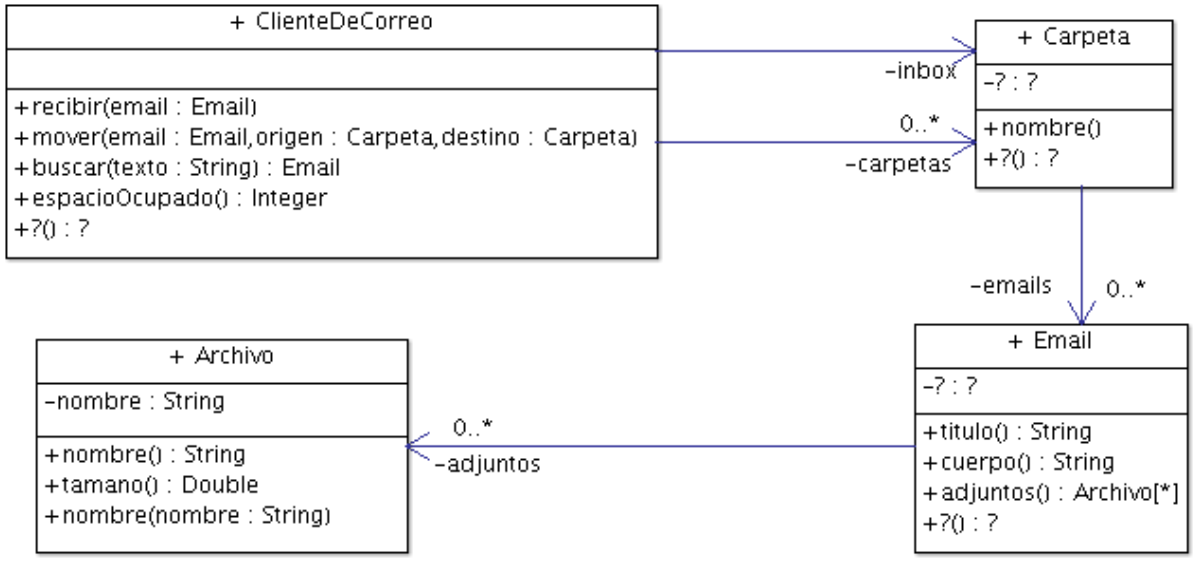
Superficie de una esfera: 4 * π * radio ^{2}
Volumen del prisma: ladoMayor * ladoMenor * altura

- π se obtiene enviando el mensaje #pi a la clase Float (Float pi)
- Para elevar un número al cuadrado, le enviamos el mensaje #squared (8 squared)
- Para elevar un número a cualquier otra potencia le enviamos el mensaje #raisedTo: (8 raisedTo: 3)

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 14. Cliente de correo con adjuntos

El diagrama de clases de UML que se muestra 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:de:a: , 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 #espacio, retorna la suma del espacio ocupado por todos los emails de todas las carpetas.
- el tamaño de un email es la suma de 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

Complete el diseño y su implementación para ofrecer la funcionalidad indicada. Ignore el hecho de que en Pharo ya existe una clase para modelar archivos.

Escriba tests de unidad para asegurar que su implementación funciona adecuadamente.

Pista: la siguiente expresión muestra como utilizar string como expresiones regulares para averiguar si un String está incluido en otro:

```
'*que*' match: 'este es un texto que dice texto' "-> true"
```

Ejercicio 15. Intervalo de tiempo

En Smalltalk, las fechas se representan con instancias de la clase Date. Por ejemplo, el mensaje de clase “#today” retorna la fecha de hoy.

- Investigue cómo hacer para crear una fecha determinada, por ejemplo 15/09/1972. Sugerencia: vea los mensajes de clase de la clase Date, en la categoría instance-creation.
- Investigue cómo hacer para ver si la fecha de hoy se encuentra entre las fechas 15/12/1972 y 15/12/2018. Sugerencia: vea los métodos de instancia que Date hereda de Magnitud (categoría comparing).
- Investigue como hacer para calcular el número de días entre dos fechas. Sugerencia: vea los métodos de instancia que Date (categoría adding)

Tarea 1
Implemente la clase DateLapse (Lapso de tiempo). Un objeto DateLapse representa el lapso de tiempo entre dos fechas. La primera fecha se conoce como “from” y la segunda como “to”. Una instancia de esta clase entiende los mensajes:

#from
“Retorna la fecha de inicio del rango”

#to
“Retorna la fecha de fin del rango”

#from: aDateFrom to: aDateTo
“Es un método privado que asigna la fecha inicial y final de un objeto DateLapse”

#sizeInDays
“retorna la cantidad de días entre la fecha 'from' y la fecha 'to'”

#includesDate: aDate
“recibe un objeto Date y retorna true si la fecha está entre el from y el to del receptor y false en caso contrario”.

Tarea 2
Escriba tests de unidad para asegurarse de que los métodos implementados funcionan adecuadamente.

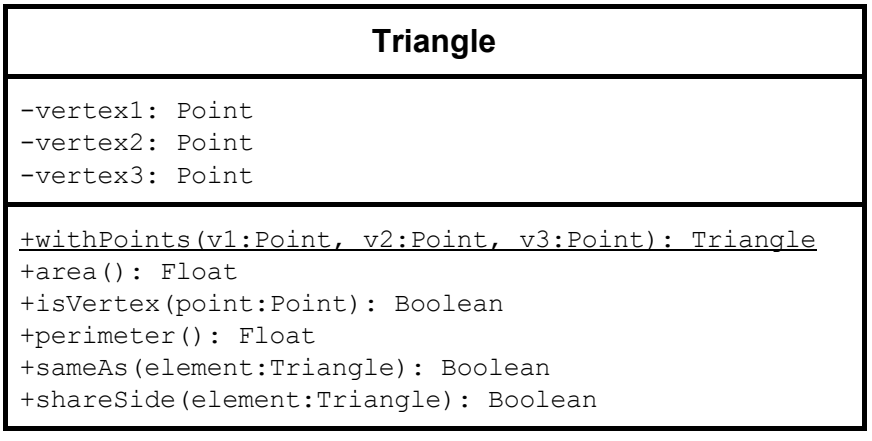
Tarea 3
Asumiendo que implementó la clase DateLapse con dos variables de instancia “from” y “to”, modifique la implementación de la clase para que su representación sea a través de los atributos “from” y “sizeInDays”. Es decir, debe basar su nueva implementación en estas variables de instancia solamente. Sugerencia: si quiere conservar la versión original, antes de realizar algún cambio haga un fileOut de la clase DateLapse.

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 ningun cambio y seguir usándolo de la misma forma. Por lo tanto, los tests que implementó en la tarea 2 deberían pasar sin problemas.

Ejercicio 16: Probando un Triángulo

Implementar (y testear) la clase Triángulo según la siguiente especificación. Estudie con detenimiento el diagrama de la clase y los puntos que siguen y cumpla las siguientes consignas:

A. Implemente la clase Triangle
B. Defina casos de prueba considerando las “particiones equivalentes” y los “valores de borde”.
C. Implement los test cases necesarios (evitando casos redundantes o repetidos) que demuestran que su implementación es correcta.



```
+isNeighborElement(element: Triangle): Boolean
```

1. En el cálculo del área se utiliza (Fórmula de Herón):
   $Area = \sqrt{sP(sP-a)(sP-b)(sP-c)}$
   en donde a, b, c son la longitud de los lados del triángulo y
   sP = $\frac{a+b+c}{2}$
2. La operación `sameAs()` retorna un booleano que dice si dos triángulos son iguales.
3. La operación `shareSide()` retorna un booleano que dice si dos triángulos tienen un lado en común
4. La operación `isNeighborElement()` retorna un booleano que dice si dos triángulos comparten al menos un vértice.

Ejercicio 17: Testeando ShiftCipher

Descargue el paquete “Cipher-Substitution” del repositorio de la materia:

```
(IceRepositoryCreator new
        url: 'https://bitbucket.org/lifia-oop/practicas-objetos-1.git';
        createRepository) updatePackage: 'Cipher-Substitution'.
```

Dicho paquete incluye la clase *ShiftCipher*. *ShiftCipher* implementa una “máquina de cifrado” por sustitución.

El cifrado por sustitución requiere dos parámetros: “clave” (key) y un alfabeto. La clave es una letra (*Character*) que indica a qué letra se mapea la primera letra del alfabeto. La clave debe ser parte del alfabeto, en caso contrario el cipher no debería crearse.

El alfabeto es un conjunto de símbolos que serán cifrados/descifrados en los mensajes.

Si un símbolo del mensaje original no se encuentra en el alfabeto de cifrado ese símbolo aparecerá en el mensaje resultante sin modificar.

En el material de teoría encontrará ejemplos sobre el uso y funcionamiento de estos cifradores.

Resuelva las siguientes preguntas:

A. Documente con un Diagrama de Clases la jerarquía donde *ShiftCipher* está definida.
B. Escriba los casos que se deberían implementar para asegurarse que la implementación de *ShiftCipher* es adecuada (hace lo que debe y no hace lo que no debe)
C. Implemente los casos de prueba con SUnit.

Comprobar Vigenère
Vigenère es un cifrado por sustitución “polialfabético”. Utiliza una “palabra” como clave y un alfabeto de referencia (como lo hace un *ShiftCipher*). En *Vigenère* cada letra de la clave se utiliza para definir un mapeo de sustitución.

Se desea comprobar que un cifrador *Vigenère* (implementado por la clase *VigenereCipher*) produce el mismo resultado que un *ShiftCipher* siempre y cuando ocurran las siguientes condiciones:

- Ambos cifradores utilizan el mismo alfabeto
- La palabra clave del *Vigenère* tiene una letra y esa letra es igual que la “clave” del *ShiftCipher*.

Resuelva las siguientes consignas:

- Escriba los casos que utilizará para la comprobación.
- Implemente dichos casos como test cases.

Ejercicio 18. Alquiler de propiedades

Necesitamos que usted implemente OOBnB, un sistema para publicar propiedades en alquiler, y para alquilarlas. Identifique objetos y responsabilidades. Diseñe e implemente. No olvide los tests de unidad. 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. 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. Si la propiedad está libre, se genera la reserva (que queda registrada en el sistema) y se retorna. La propiedad pasa a estar ocupada en esas fechas. Si la propiedad no está libre no hace nada.
Calcular el precio de una reserva: dada una reserva, obtener el precio a partir del precio por día de la propiedad y la cantidad de noches de la reserva.

Cancelar una reserva: Si la fecha de inicio de la reserva es posterior a la fecha actual (Date today) 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.

Ejercicio 19. Políticas de cancelación

En un sistema de alquiler de propiedades del ejercicio 16 (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 una semana antes y 50% si se hace 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 20. 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.

- **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 una 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 una 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 hora de comienzo, la duración en minutos, el número del teléfono que llama y el del teléfono que recibe. El sistema guarda registro de la llamada. El sistema retorna el registro de la llamada.
- **Registrar una llamada interurbana**. Se provee la 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 registro de la llamada. El sistema retorna el registro de la llamada.
- **Registrar una llamada internacional**. Se provee la hora de comienzo, la duración en minutos, el número del teléfono que llama y el del teléfono que recibe. Se provee país de origen y país destino de la llamada. El sistema guarda 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 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 100km, $2.5 para distancias entre 100km y 500km, y $3 para distancias de mas de 500km)
3. Las llamadas internacionales tienen un costo por minuto que depende del país destino y de la hora (el precio diurno de 8:00 a 20:00 es un valor, y precio nocturno de 20:00 a 8:00 es otro). Por ahora utilice $3 como precio diurno para todos los países y $4 como precio diurno 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 Smalltalk 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 que pasaría si al facturar se provee (como un parámetro mas) el "cuadro tarifario". ¿Como sería ese objeto? ¿que responsabilidad le podría delegar? ¿Como haríamos para tener montos diferentes para los distintos países en las llamadas internacionales?

Ejercicio 21. 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 el empleado con ese CUIL o nil 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, y la fecha de fin (si corresponde). 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 para el empleado.

Obtener empleados con contratos vencidos. El sistema devuelve la lista de todos aquellos empleados cuyos contratos hayan finalizado al día de hoy.

Generar recibos de cobro. Por cada empleado (con contrato 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 fué 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, el monto acordado por tener cónyuge a cargo (si es que tiene cónyuge a cargo), y 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 cuanto se alcanza cierta antigüedad, en función esta escala: 5 años 30%, 10 años 50%, 15 años 70%, 20 años 100%

Su tarea es diseñar y programar en Pharo Smalltalk 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 Smalltalk de la funcionalidad requerida.
3. Implementar los tests (SUnit) que considere necesarios.

Ejercicio 22. Mercado 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 nil.
- **Registrar un cliente:** Se indica el nombre del cliente y su dirección. Se agrega cliente y se lo retorna.
- **Buscar un cliente:** Se indica el nombre del cliente que se desea buscar/recuperar. Si existe lo retorna. Si no retorna nil.
- **Poner un producto a la venta:** Se indica el nombre del producto, su descripción, su precio, la cantidad de unidades disponibles y el vendedor.
- **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.
- **Obtener las formas de pago disponibles para un producto.**
- **Cambiar (agregar o quitar) las formas de pago disponibles para un producto.**
- **Obtener los mecanismos de envío disponibles para un producto**
- **Cambiar (agregar o quitar) los mecanismos de envío disponibles para un producto.**

- **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 posible* 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 la del cliente. Asuma que existe una clase Mapa, cuyas instancias entienden el mensaje #distanciaEntre:y: 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).

El sistema permitiría por ejemplo,

1. registrar el vendedor "Todo lindo" con dirección "Calle 16, Nº90, La Plata"
2. registrar al cliente "Juan Delos Palotes" con dirección "Mitre 3587, Cañuelas".
3. poner a la venta por “Todo lindo” el producto "MyPhone 3", con descripción "El teléfono que siempre quisiste", precio $8600, 40 unidades disponibles.
4. agregar "en 6 cuotas" como forma de pago adicional a ese producto
5. agregar "express a domicilio" como mecanismo de envío disponible para ese producto (notar que no se podrá "retirar en sucursal del correo".
6. crear un pedido de Juan, de 2 unidades del MyPhone. Para que se lo envíen "express a su domicilio".
7. calcular el costo del pedido que hizo Juan.
8. ver cuántos MyPhones quedan disponibles.

Su tarea es diseñar y programar en Pharo Smalltalk 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 Smalltalk de la funcionalidad requerida.
3. Implementar tests (SUnit) para el cálculo del costo de un pedido.

4. Expresiones Smalltalk (Playground) que demuestren cómo con su código se implementaría el ejemplo provisto más arriba.

Ejercicio 23

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

En breve...