Trabajo Práctico 3
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación — 1er cuat. 2015

Fecha de entrega: 23 de junio

Introducción

El objetivo del trabajo será implementar un modelo para la registración de mediciones: números con sus respectivas unidades (e.g., `10 metros`). En este modelo se deberá poder realizar operaciones entre mediciones y también con escalares, para lo cual no sólo deberá actualizarse las magnitudes, sino también las unidades resultantes o, en caso de que la operación no tenga sentido, que se genere el error correspondiente.

Con la implementación se debe modelar, por ejemplo, las siguientes operaciones con medidas y escalares:

$$5\,\mathrm{m/s} * 4\,\mathrm{m} = 20\,\mathrm{m}^2/\mathrm{s} \quad , \quad \frac{12\,\mathrm{kg}}{10\,\mathrm{s}} = 1.2\,\mathrm{kg/s} \quad , \quad 10 * 2\,\mathrm{m} = 20\,\mathrm{m}$$

El trabajo estará guiado a través de una serie de tests inspirados en la técnica *Test Driven Development (TDD)*[1].

Por consiguiente, se deberá implementar el mínimo código razonable para lograr ir pasando los tests de manera progresiva, manteniendo en funcionamiento los tests previos. Cada ejercicio irá acompañado de un test que tenga la información necesaria para resolverlo y algunas sugerencias para su implementación.

Ejercicios

Ejercicio 1

Implementar las clases y métodos básicos requeridos para la creación de unidades. Para ello se debe seguir la definición de `Test01CrearUnidades`, donde se pide que cada unidad tenga un nombre, y los escalares también tengan una unidad.

Ejercicio 2

Implementar las clases y métodos básicos requeridos para la creación de mediciones. Para ello se debe seguir las definiciones del `Test02CrearMedidas`, que requieren la modificación de la clase *Number*.

[1] Tomar la metodología presentada como una introducción a la técnica. Para más detalles, recomendamos la materia POO de Hernán Wilkinson.

Ejercicio 3

Implementar la igualdad de mediciones, tomando en cuenta el valor y la unidad, según lo definido en `Test03IgualdadMedidas`. En este test se hace uso del operador de desigualdad ($\sim=$), tener en cuenta que el mismo utiliza internamente el operador = y no hay que definirlo explícitamente.

Ejercicio 4

Implementar las operaciones de las mediciones con escalares, como se valida con el test `Test04OperacionesConEscalares`.
Observar que en este test se utiliza la técnica de *double dispatch* cuando se ejecuta la siguiente expresión: `4 * 4 metro`. Aquí, se envía el mensaje `*` con la medida `4 metro` a la instancia `4` de la clase `Integer`. En el conjunto de colaboraciones de `Integer>>* param` "se envía de cierta forma" el mensaje `* self` al parámetro recibido `param` (`4 metro`), delegando de esta manera la responsabilidad de cómo realizar dicha operación a la clase `Medida`.

Ejercicio 5

Implementar las clases y mensajes necesarios para la multiplicación de unidades siguiendo todas las restricciones del `Test05ProductoDeUnidades`. Nuevamente aquí se debe utilizar *double dispatch* para resolver el producto entre unidades.
Por ejemplo, durante la ejecución de la expresión `Unidad metro * Unidad escalar` el método `UnidadBasica>>* param` deberá delegar la responsabilidad al parámetro. Es decir, en el cuerpo del método `UnidadBasica>>* param` se realizará la siguiente colaboración `param productoBasica: self`. De esta manera, el método `UnidadEscalar>>productoBasica: param` retornará la unidad tomada cómo parámetro (en este caso, `Unidad metro`).

Ejercicio 6

Implementar la operación de producto de medidas, que contemple a las cantidades tanto como a las unidades, como se describe en `Test06ProductoDeMedidas`.

Ejercicio 7

Implementar los mensajes necesarios para la implementación de sumas y restas de unidades, de acuerdo a lo definido por el `Test07SumaDeUnidades`.

Ejercicio 8

Implementar la impresión de las unidades, de acuerdo a los requerimientos definidos en `Test08ImprimirUnidades`.

Ejercicio 9

Implementar la impresión de medidas, contemplando a las cantidades tanto como a las unidades, como se define en `Test09ImprimirMedidas`.

Ejercicios opcionales (pero muy gratificantes)

Ejercicio 10

Implementar las clases y mensajes necesarios para la división de unidades siguiendo todas las restricciones en `Test10DivisionDeUnidades`. Aquí se deberá utilizar *double dispatch*, como se hizo con el producto.

Ejercicio 11

Implementar la operación de división de medidas, que contemple las cantidades tanto como las unidades, como se describe en `Test11DivisionDeMedidas`.

Notar que al evaluar `2 / 1 segundo` eventualmente se envía el mensaje `Medida>>divInv: 2`. Esto es por la implementación dada de `Medida>>adaptToNumber: anInteger andSend: aString`.

Ejercicio 12

Implementar la impresión de la división de unidades, de acuerdo a los requerimientos definidos en `Test12ImprimirUnidades`.

Ejercicio 13

Implementar la impresión de medidas con división de unidades, contemplando tanto cantidad como las unidades, como se define en `Test13ImprimirMedidas`.

Ejercicio 14

Un renombre de unidades permite denotar un cierto conjunto de unidades de una manera compacta. Por ejemplo, $\mathrm{N} = \mathrm{m\,kg\,/s^2}$ (Newton) y $\mathrm{Pa} = \mathrm{N/m^2}$ (Pascal). Implementar el renombre de unidades, como se plantea en `Test14RenombreUnidades`.

La idea de la implementación es agregar a `Unidad` un método por renombre que se desea contemplar. Por ejemplo, `Unidad>>newton` que devuelva una instancia de `Unidad` usando las operaciones definidas anteriormente: `(Unidad metro * Unidad kilogramo) / (Unidad segundo * Unidad segundo)`.

En el método `Unidad>>renombres` se espera usar *reflection* para listar todos los métodos de `Unidad` salvo renombres, y si el resultado no es ni la unidad escalar ni una unidad básica, entonces se entiende que es un renombre. Luego, el método renombres devuelve un diccionario de `String` a `Unidad` donde la clave es el renombre a usar. Usar el mensaje `Unidad class>>selectors` para obtener los métodos de una clase.

```
Unidad class selectors do: [ :sel |
  (sel = #renombres) ifFalse: [ | unidad |
    unidad := Unidad perform: sel.
    ...
  ]
]
```

Luego, se deberá adaptar el `DivisionDeUnidades>>printOn` para aprovechar los renombres.

Pautas de entrega

El entregable debe contener:

- un archivo `.st` con todas las clases implementadas
- versión impresa (legible) del código, comentado adecuadamente
- **NO** hace falta entregar un informe sobre el trabajo

Se espera que el diseño presentado tenga en cuenta los siguientes factores:

- definición adecuada de clases y subclases, con responsabilidades bien distribuidas
- uso de polimorfismo para evitar exceso de condicionales
- intento de eliminar código repetido utilizando las abstracciones que correspondan

Consulten todo lo que sea necesario.

Consejos y sugerencias generales

- Lean al menos el primer capítulo de *Pharo by example*, en donde se hace una presentación del entorno de desarrollo.
- Explorar la imagen de Pharo suele ser la mejor forma de encontrar lo que uno quiere hacer. En particular tengan en cuenta el buscador (`shift+enter`) para ubicar tanto métodos como clases.
- No se pueden modificar los test entregados, aunque pueden agregar tests propios (se recomienda agregar lo que utilicen para hacer sus propias pruebas).

Importación y exportación de paquetes

En Pharo se puede importar un paquete arrastrando el archivo del paquete hacia el intérprete y seleccionando la opción "`FileIn entire file`". Otra forma de hacerlo es desde el "`File Browser`" (botón derecho en el intérprete > `Tools` > `File Browser`, buscar el directorio, botón derecho en el nombre del archivo y elegir "`FileIn entire file`").

Para exportar un paquete, abrir el "`System Browser`", seleccionar el paquete deseado en el primer panel, hacer click con el botón derecho y elegir la opción "`FileOut`". El paquete exportado se guardará en el directorio `Contents/Resources` de la instalación de Pharo (o en donde esté la imágen actualmente en uso).