Práctica 4

Herencia, interfaces y excepciones

Inicio: Semana del 16 de marzo.
Duración: 3 semanas.
Entrega: Semana del 13 de abril
Peso de la práctica: 30%

El objetivo de esta práctica es el diseño de una librería para el manejo de unidades de medida utilizando técnicas de programación orientada a objetos más avanzadas que en las prácticas precedentes, como son la herencia, la ligadura dinámica, las excepciones y las *interfaces*. Se trata de servirse de estas técnicas para reducir código redundante, obtener una API fácilmente extensible, y desarrollar un programa bien estructurado que refleje de una forma directa los conceptos del dominio del problema.

En el desarrollo de esta práctica se utilizarán principalmente los siguientes conceptos de Java y de programación orientada a objetos:

- *Herencia y ligadura dinámica*
- *Manejo de excepciones*
- *Uso de interfaces Java*
- *Estructuración en paquetes*

Motivación

Todos los lenguajes de programación cuentan con diversos tipos de datos para el almacenamiento de valores numéricos (como `double`, `int`, `float`, etc). No obstante, las cantidades almacenadas pueden representar magnitudes físicas (de longitud, tiempo, masa), que utilizan un sistema métrico determinado. Por ejemplo el Sistema Internacional define una serie de unidades de medida para cada tipo de magnitud (metros, kilómetros, etc para medir longitudes; segundos, horas, etc para medir tiempo). No obstante, existen varios sistemas métricos diferentes en uso hoy en día. Por ejemplo el sistema Imperial (muy usado en el Reino Unido) prescribe las yardas, millas, etc como unidades para medir la longitud.

El problema puede surgir si un programa utiliza magnitudes expresadas en unidades de distintos sistemas métricos, y no se realiza correctamente la conversión correspondiente. De hecho, este es un error típico que ha dado lugar a numerosos fallos de software. Por ejemplo el satélite "*Mars Climate Orbiter*" de la NASA (mostrado en la Figura 1 en fase de pruebas) siguió una trayectoria errónea debida a que la base terrestre enviaba datos en unidades que no estaban en el sistema internacional, cosa que esperaba el programa que controlaba el satélite [1]. El resultado fue un fallo en su comportamiento, que hizo que se desintegrara en la atmósfera del planeta Marte en Septiembre de 1999. El coste de esta misión fallida fue de 397.6 millones de dólares.

Figura 1: La Mars Climate Orbiter en fase de pruebas

El objetivo de esta práctica es diseñar una librería extensible que permita describir sistemas métricos, unidades de medida, y conversiones entre ellos, de tal forma que se puedan construir programas seguros, que eviten errores de conversiones. La idea es poder declarar las unidades de medida, y que las conversiones se realicen automáticamente. Debido a restricciones de tiempo, no desarrollaremos la librería en toda su extensión, sino que haremos varias simplificaciones y limitaremos su ámbito.

[1] https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

Conceptos y estructura de la librería

Antes de empezar la práctica, analizaremos los conceptos a manejar. En los distintos apartados, se te proporcionarán *interfaces* Java para algunos de los elementos de la librería. El objetivo de usar interfaces es permitir implementaciones que sean más flexibles y extensibles. Pese a que se te da una guía detallada, a diferencia de las prácticas anteriores, te dejamos más libertad para realizar el diseño que creas más conveniente. Durante la práctica, también debes idear una estructura adecuada de paquetes para la librería.

La librería debe manejar unidades físicas (PhysicalUnit), tales como los metros o las yardas. Dichas unidades miden cierta cantidad (Quantity) de longitud, tiempo o masa, entre muchas otras. Por simplificar, asumiremos sólo dos tipos de cantidad: de longitud (Length) y de tiempo (Time). Un sistema métrico (tal como el sistema internacional, o el imperial) para una determinada cantidad (por ejemplo longitud), engloba una serie de unidades físicas, siendo una de ellas la *unidad base (o patrón)*. Por ejemplo, el sistema internacional de longitud incluye los milímetros, centímetros, decímetros, metros, decámetros, hectómetros y kilómetros, entre otras. La medida base de este sistema es el metro. Por simplificar, consideraremos el sistema métrico internacional para tiempo y longitud, y el sistema imperial para longitud.

Una magnitud (Magnitude) está formada por un valor y unas unidades físicas. Por ejemplo 5 metros, o 7 segundos. La idea es asignar a esta clase responsabilidades como la conversión a otra unidad, o la suma/resta de otras magnitudes, con la consecuente transformación de unidades.

Apartado 1: Unidades Físicas (simplificadas) y Excepciones (3 puntos)

En este apartado implementarás (parcialmente) las unidades físicas. Para abreviar, consideraremos 3 unidades de longitud (milímetro, metro y kilómetro) y 3 de tiempo (milisegundo, segundo y hora). Ambas pertenecen al sistema internacional. La siguiente interfaz muestra la funcionalidad mínima esperable de una unidad física.

```
public interface IPhysicalUnit {
        boolean  canTransformTo(IPhysicalUnit u);
        double   transformTo(double d, IPhysicalUnit u) throws QuantityException;
        Quantity getQuantity();
        String   abbrev();
        IMetricSystem getMetricSystem();  // No implementar de momento en este apartado
}
```

El método canTransformTo devuelve si la unidad se puede transformar a otra que se le pasa como parámetro. Por ejemplo, un metro se puede convertir a kilómetro (y viceversa), pero un metro no se puede convertir a segundo (es una unidad de distinto Quantity). El método transformTo convierte una magnitud a una unidad que se le pasa como parámetro. El método puede lanzar excepciones debido a que las Quantity de las unidades son distintas (p.ej., tiempo y longitud), o porque la unidad de destino es desconocida (por ejemplo, es una unidad de un sistema métrico distinto). En el apartado 5 diseñaremos una manera de transformar entre sistemas métricos diferentes. Por el momento, realiza una implementación sencilla de un sistema métrico que simplemente contenga una serie de unidades, de una determinada Quantity. Por simplicidad puedes implementar Quantity como un enumerado. Finalmente abbrev devuelve un String que representa la unidad física de manera abreviada (por ejemplo "m" para metros, o "km" para Kilómetros).

Debes procurar que el método transformTo sea lo más general y conciso posible. Es decir, sería mala práctica de programación tener una implementación distinta por cada combinación de unidades. Igualmente, debes evitar que se puedan crear instancias de las distintas unidades de manera externa a la librería (es decir, debe existir un único objeto KILOMETER, METER, etc).

A modo de ejemplo, el programa de más abajo debería producir la salida de más abajo.

```
public class PhysicalUnitTest {
  public static void main(String[] args) throws QuantityException {
        IPhysicalUnit meter = SiLengthMetricSystem.METER;
        System.out.println(meter);                                  // This is how a meter is printed (abrev + Quantity)
        System.out.println(meter.canTransformTo(SiLengthMetricSystem.KILOMETER));   // Yes, we can
        System.out.println(meter.canTransformTo(SiTimeMetricSystem.SECOND));        // No, we don't
        System.out.println("1000 m en km: "+meter.transformTo(1000, SiLengthMetricSystem.KILOMETER));
        try {
                System.out.println("1000 m en s: "+meter.transformTo(1000, SiTimeMetricSystem.SECOND)); // Exception!
        } catch (QuantityException e) {
                System.out.println(e);
        }
  }
}
```

Salida esperada:

```
m L
true
false
1000 m en km: 1.0
Quantities t and L are not compatible
```

De momento, ignora el método getMetricSystem, así como la interfaz IMetricSystem.

Apartado 2: Unidades Físicas y Sistemas Métricos (1,5 puntos)

En este apartado, definirás con más detalle la estructura de un sistema métrico, sobre la base de la siguiente interfaz.

```
public interface IMetricSystem {
        IPhysicalUnit base();
        Collection<IPhysicalUnit> units();
}
```

Como ves, un sistema métrico tiene una colección de unidades, y una unidad base. Debes implementar al menos 3 unidades del sistema métrico internacional de longitud y tiempo, y al menos 3 del sistema imperial de longitud. De momento, sigue considerando transformaciones sólo entre unidades del mismo sistema. Debes procurar que estas tres implementaciones no tengan código redundante, mediante un buen uso de conceptos de orientación a objetos.

Debes considerar algún mecanismo que evite crear instancias de sistemas métricos de manera externa. En el ejemplo de más abajo, los sistemas métricos tienen una constante SYSTEM, que almacena el único objeto del sistema (esto es, es un patrón *Singleton*). El resultado de ejecutar el programa de prueba da el resultado de más abajo.

```
public class MetricSystemTest {

  public static void main(String[] args) {
        IMetricSystem ms = SiLengthMetricSystem.SYSTEM;
        //new SiLengthMetricSystem();    // compilation error
        System.out.println(ms.units());
        System.out.println("Base = "+ms.base());

        System.out.println(SiLengthMetricSystem.METER.canTransformTo(ImperialLengthMetricSystem.FOOT));
        // No: different metric systems
  }
}
```

Salida esperada:

```
[km L, m L, mm L]
Base = m L
false
```

Apartado 3: Magnitudes, operaciones y conversiones entre el mismo sistema (2 puntos)

En este apartado, crearemos soporte para las magnitudes, que están formadas por un valor numérico y una unidad. Las magnitudes tendrán métodos para añadir y restar otra magnitud. Dichos métodos transformarán la magnitud que se pasa como parámetro a la unidad de la magnitud que recibe la invocación, efectuarán la operación, modificando el objeto y devolviéndolo (para que se puedan concatenar operaciones). Debe además incluir un método para transformar la magnitud a otra unidad, así como para obtener la unidad de la magnitud y su valor. Los métodos deben tener en cuenta posibles errores, que se señalarán mediante excepciones compatibles con QuantityException.

```
public interface IMagnitude {
        IMagnitude add (IMagnitude m) throws QuantityException;
        IMagnitude subs(IMagnitude m) throws QuantityException;
        IMagnitude transformTo(IPhysicalUnit c) throws QuantityException;
        IPhysicalUnit getUnit();
        double getValue();
}
```

A modo de ejemplo, el siguiente programa resulta en la salida de más abajo. Como ves, la clase Magnitude implementa la interfaz IMagnitude.

```
public class MainTest {
  public static void main(String[] args) throws QuantityException{
        IMagnitude m  = new Magnitude(12.5, SiLengthMetricSystem.KILOMETER);
        Magnitude m2 = new Magnitude(12.5, SiLengthMetricSystem.METER);

        System.out.println(m2.add(m));                  // m converted to meters and added to m2
        System.out.println(m.subs(m2).add(m2));         // operations can be chained
        System.out.println(m.transformTo(SiLengthMetricSystem.METER));

        Magnitude s1 = new Magnitude(65, SiTimeMetricSystem.SECOND);

        try {
                System.out.println(s1.add(m));
        } catch (QuantityException q) {
                System.out.println(q);
        }
  }
}
```

Salida esperada:

```
12512.5 [m L]
12.5 [km L]
12500.0 [m L]
Quantities t and L are not compatible
```

Nótese que sólo debes implementar la suma y la resta. La multiplicación y división resultan en general en unidades compuestas (por ejemplo m / s), y ese caso se dejará para el apartado opcional (aunque de manera simplificada).

Apartado 4: Conversiones entre distintos sistemas (3 puntos)

En este apartado, implementarás conversiones entre distintos sistemas métricos. Como prueba de concepto, sólo se pide convertir entre el sistema internacional y el imperial de longitud. Para afrontar esta parte, una posible estrategia de diseño es crear clases conversoras por cada par de sistemas. La siguiente interfaz te muestra la funcionalidad esperada de una clase conversora.

```
public interface IMetricSystemConverter {
        IMetricSystem sourceSystem();
        IMetricSystem targetSystem();
        IMagnitude transformTo(IMagnitude from, IPhysicalUnit to) throws UnknownUnitException;
        IMetricSystemConverter reverse();
}
```

Como ves, se pueden obtener el Sistema fuente y destino, y existe un método de conversión de una magnitud del sistema fuente a una unidad del sistema destino. Debes hacer las comprobaciones correspondientes y lanzar excepciones en caso de error (por ejemplo si la unidad destino no es parte del sistema destino). Se recomienda integrar las excepciones necesarias en la jerarquía de excepciones que ya debes haber creado en apartados anteriores.

El método reverse devuelve un conversor entre el sistema destino y el fuente. Una implementación de esta interfaz típicamente utilizará un multiplicador para convertir de la unidad base del sistema fuente al destino. Por ejemplo, para convertir del sistema internacional al imperial, usaremos que un metro son 3.280839895 pies. El conversor devuelto por el método reverse simplemente utilizará el inverso (recíproco) de este multiplicador.

Para mejorar la usabilidad de la librería, cada sistema métrico tendrá un registro, donde se puedan añadir los conversores. De esta manera, debes extender la interfaz de IMetricSystem con un método para obtener el conversor a un sistema métrico determinado.

```
public interface IMetricSystem {
        IPhysicalUnit base();
        Collection<IPhysicalUnit> units();
        // Added for convertible
        IMetricSystemConverter getConverter(IMetricSystem to);
}
```

Como ves en el siguiente ejemplo, por defecto no es posible convertir de kilómetros a millas. Cuando se registra un conversor en `SiLengthMetricSystem`, ya sí debe resultar posible realizar dicha conversión. Al registrar un conversor, se registra su inverso también. No olvides añadir las nuevas excepciones que sean necesarias.

```
public class ConversionTest {
  public static void main(String[] args) throws QuantityException {
        Magnitude m  = new Magnitude(10, SiLengthMetricSystem.KILOMETER);

        IMagnitude enMillas = null;

        try {
                enMillas = m.transformTo(ImperialLengthMetricSystem.MILE);
        } catch (QuantityException e) {
                System.out.println(e);
        }
        SiLengthMetricSystem.registerConverter(new SiLength2ImperialConverter());
        // Registers the converter and the reverse
        enMillas = m.transformTo(ImperialLengthMetricSystem.MILE);

        System.out.println("En millas = "+enMillas);
        System.out.println("En m = "+ enMillas.transformTo(SiLengthMetricSystem.METER));
  }
}
```

Salida esperada:

```
Cannot transform km L to ml L
En millas = 6.213711922348486 [ml L]
En m = 10000.0 [m L]
```

(*) Nota: es posible que obtengas valores ligeramente distintos debido a redondeos

Apartado 5: Diagrama de clases y explicación del diseño (0,5 puntos)

a) Debes entregar un diagrama de clases que muestre el diseño efectuado. No olvides que el objetivo del diagrama es explicar el diseño con un nivel de abstracción adecuado, no es "*Java en dibujos*". De esta manera, debes obviar constructores, getters y setters, y debes representar las colecciones, arrays y referencias usadas como asociaciones del tipo más adecuado. No olvides incluir las interfaces que has usado, y las relaciones de implementación entre clases e interfaces. Incluye una pequeña explicación del diseño, así como de las decisiones que has tomado.

b) ¿Es extensible tu diseño? Indica qué pasos habría que dar para:

b.1) añadir nuevas unidades a un sistema existente

b.2) añadir la cantidad Masa al Sistema Métrico Internacional

c) ¿Qué desventajas o limitaciones tiene el diseño de la librería?

Apartado 6 (Opcional): Unidades compuestas (1 punto)

En este apartado, deberás implementar unidades compuestas, por ejemplo m/s. Para ello, una opción es extender la interfaz IPhysicalUnit de la siguiente manera:

```
public interface ICompositeUnit extends IPhysicalUnit{
        Operator getOperator();
        IPhysicalUnit getLeftUnit();
        IPhysicalUnit getRightUnit();
}
```

Donde Operator es un enumerado o una clase que representa un operador aritmético binario (considera por simplicidad sólo * y /), getLeftUnit y getRightUnit devuelven la unidad izquierda y derecha (que pueden ser a su vez unidades simples o compuestas). Una unidad compuesta se puede convertir a otra (método canTransformTo) si las unidades izquierdas y derechas se pueden traducir. Para convertir una magnitud d entre unidades compuestas, se multiplica d al resultado de operar (con * o /, según sea el operador de la unidad compuesta) los resultados de convertir 1.0 a las unidades izquierda y derecha. Es decir, si queremos convertir 10 m/s a ml/h (millas/hora), tenemos que:

1.0 m = 0.000621371 ml, y que 1.0 s = 0.000277778 h, con lo que 10 m/s = 10 * 0.000621371/0.000277778 = 22,3693381 ml/h.

Ten en cuenta que las unidades compuestas implican además la implementación de Quantity compuestas.

A modo de ejemplo, el siguiente programa debe dar la salida de más abajo.

```
public class CompositeTest {
  public static void main(String[] args) throws QuantityException{
        SiLengthMetricSystem.registerConverter(new SiLength2ImperialConverter());

        Magnitude velocSI = new Magnitude(10, new CompositeUnit( SiLengthMetricSystem.METER,
                                                                 Operator.DIV,
                                                                 SiTimeMetricSystem.SECOND));
        Magnitude velocImp = new Magnitude(0, new CompositeUnit( ImperialLengthMetricSystem.MILE,
                                                                 Operator.DIV,
                                                                 SiTimeMetricSystem.HOUR));
        Magnitude velocSI2 = new Magnitude(0, new CompositeUnit( SiLengthMetricSystem.KILOMETER,
                                                                 Operator.DIV,
                                                                 SiTimeMetricSystem.HOUR));
        System.out.println(velocSI);
        System.out.println(velocImp);
        System.out.println(velocImp.add(velocSI));       // implica convertir m/s a millas/hora
        System.out.println(velocSI2.add(velocSI));       // implica convertir m/s a km/hora
  }
}
```

Salida esperada

```
10.0 [m / s]
0.0 [ml / h]
22.369362920454545 [ml / h]
36.0 [km / h]
```

Normas de Entrega:

Se deberá entregar

- un directorio **src** con todo el código Java de todos los apartados, incluidos los datos de prueba y testers adicionales que hayas desarrollado en los apartados que lo requieren
- un directorio **doc** con la documentación generada
- un archivo PDF con el apartado 5.
- Si has hecho el apartado optativo, entrégalo en otro proyecto o directorio.

Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2213, entregarían el fichero: GR2213_MarisaPedro.zip.