Assignment 4

Inheritance, Interfaces and Exceptions

Inicio: Week of March 16th
Duration: 3 weeks
Delivery: Week of April 13th
Weight: 30%

The objective of this assignment is the design of a library for the management of *measurement units* using more advanced object-oriented programming techniques than in previous labs, such as inheritance, dynamic binding, exceptions and interfaces. The objective is to use these techniques to reduce redundant code, obtain an easily extensible API, and develop a well-structured program that directly reflects the concepts of problem domain.

The following concepts of Java and object-oriented programming will mainly be used in the development of this assignment:

- *Inheritance and dynamic binding*
- *Exception handling*
- *Use of Java interfaces*
- *Structuring the code into packages*

Motivation

All programming languages have different data types for storing numerical values (such as `double`, `int`, `float`, etc.). However, the stored quantities can represent physical quantities (of length, time, mass), which use a specific metric system. For example, the International System defines a series of units of measure for each type of magnitude (meters, kilometers, etc. to measure lengths; seconds, hours, etc. to measure time). However, there are several different metric systems in use today. For example, the Imperial system (widely used in the United Kingdom) prescribes yards, miles, etc. as units for measuring length.

The problem may arise if a program uses quantities expressed in units of different metric systems, and the corresponding conversion is not performed correctly. In fact, this is a typical error that has resulted in numerous software failures. For example, the NASA "*Mars Climate Orbiter*" satellite (shown in Figure 1 in the testing phase) followed an erroneous trajectory because the ground base sent data in units that were not in the international system, which the controller program expected [1]. The result was a failure in their behavior, which caused it to disintegrate in the atmosphere of planet Mars in September 1999. The cost of this failed mission was $397.6 million.

Figure 1: Mars Climate Orbiter being tested

The objective of this practice is to design an extensible library that allows to describe metric systems, units of measurement, and conversions between them, so that safe programs can be built, which avoid conversion errors. The idea is to be able to declare the units of measure, and that the conversions are carried out automatically. Due to time constraints, we will not develop the library in full, but we will make several simplifications and limit its scope.

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

Concepts and structure of the library

Before starting this lab, we will analyse the main concepts you need to handle. In the different sections, you will be provided with Java *interfaces* for some of the library elements. The goal of using interfaces is to allow implementations that are more flexible and extensible. Although you are given a detailed guide, unlike previous assignments, you are given more freedom to make the design you believe is most convenient. During the assignment, you should also devise an appropriate package structure for the library.

The library must handle *physical units* (PhysicalUnit), such as meters or yards. These units measure a certain *quantity* (Quantities) of length, time or mass, among many others. For simplicity, we will assume only two types of quantity: Length and Time. A metric system (such as the international system, or the imperial one) for a certain quantity (for example length), encompasses a series of physical units, one of them being the *base unit* (or pattern). For example, the international system of length includes millimeters, centimeters, decimeters, meters, decameters, hectometers and kilometers, among others. The base measure of this system is the meter. For simplicity, we will consider the international metric system for time and length, and the imperial system for length.

A Magnitude is formed by a value and physical units. For example 5 meters, or 7 seconds. The idea is to assign to this class responsibilities such as conversion to another unit, or the addition/subtraction of other quantities, with the consequent transformation of units.

Section 1: (Simplied) Physical Units and Exceptions (3 points)

In this section you will (partially) implement the infrastructure for physical units. In short, we will consider 3 units of length (millimeter, meter and kilometer) and 3 units of time (millisecond, second and hour). Both belong to the international system. The following interface shows the minimum expected functionality of a physical unit.

```
public interface IPhysicalUnit {
   boolean  canTransformTo(IPhysicalUnit u);
   double   transformTo(double d, IPhysicalUnit u) throws QuantityException;
   Quantity getQuantity();
   String   abbrev();
   IMetricSystem getMetricSystem();  // for the moment, do not implement in this part
}
```

The canTransformTo method returns if the unit can be transformed to another one that is passed as a parameter. For example, a meter can be converted to kilometer (and vice versa), but a meter cannot be converted to second (it is a unit of a different Quantity). The transformTo method converts a magnitude to a unit that is passed as a parameter. The method may throw exceptions because the Quantity of the units are different (eg, time and length), or because the destination unit is unknown (for example, it is a unit of a different metric system). In section 5 we will design a way to transform between different metric systems. At the moment, please make a simple implementation of a metric system, containing just a series of units, of a certain Quantity. For simplicity you can implement Quantity as an enumeration. Finally, abbrev returns a String that represents the physical unit in an abbreviated way (for example "m" for meters, or "km" for Kilometers).

You must ensure that the transformTo method is as general and concise as possible. That is, it would be bad programming practice to have a different implementation for each combination of units. Likewise, you should avoid that instances of the different units can be created externally to the library (that is, there must be a single KILOMETER, METER, etc. object).

As an example, the following program should produce the output below.

```
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 in km: "+meter.transformTo(1000, SiLengthMetricSystem.KILOMETER));
       try {
              System.out.println("1000 m in s: "+meter.transformTo(1000, SiTimeMetricSystem.SECOND)); // Exception!
       } catch (QuantityException e) {
              System.out.println(e);
       }
  }
}
```

Expected output:

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

For the moment, please ignore method getMetricSystem, as well as the interface IMetricSystem.

Section 2: Physical Units and Metric Systems (1,5 points)

In this section, you will define in more detail the structure of a metric system, based on the following interface:

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

As you can see, a metric system has a collection of units, and a base unit. You must implement at least 3 units of the international metric system of length and time, and at least 3 of the imperial system of length. At the moment, please consider only transformations between units of the same system. You must ensure that these three implementations do not have redundant code, based on a good use of object-oriented concepts.

You should consider some mechanism that avoids creating instances of metric systems in an external way. In the example code below, the metric systems have a constant SYSTEM, which stores the unique object (that is, it is a Singleton pattern). The result of executing the following program produces the result shown below.

```
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
  }
}
```

Expected output:

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

Section 3: Magnitudes, operations and conversions between the same system (2 points)

In this section, we will create support for the magnitudes, made of a numerical value and a unit. The magnitudes will have methods to add and subtract another magnitude. These methods will first transform the magnitude that is passed as a parameter to the unit of the magnitude that receives the invocation. Then, the methods will perform the addition or subtratcion operation, modifying the object and returning it (so that operations can be concatenated). The interface must also include a method to transform the magnitude to another unit, as well as to obtain the unit of the magnitude and its value. The methods must take into account possible errors, which will be signaled by exceptions compatible with 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();
}
```

As an example, the following program results in the output below. As you can see, the Magnitude class implements the IMagnitude interface.

```
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);
       }
  }
}
```

Expected output:

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

Please note that you should only implement addition and subtraction. Multiplication and division generally result in compound units (for example m / s), and that case will be left for the optional section (although in a simplified way).

Section 4: Conversions between different systems (3 points)

In this section, you will implement conversions between different metric systems. As a proof of concept, it is only requested to convert between the international and the imperial system in length. To address this part, a possible design strategy is to create converter classes for each pair of systems. The following interface shows the expected functionality of a converter class.

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

As you can see, the source and destination systems can be obtained, and there is a method for converting a magnitude from the source system to a unit of the destination system. You must make the corresponding checks and throw exceptions in case of error (for example if the destination unit is not part of the destination system). It is recommended to integrate the necessary exceptions in the hierarchy of exceptions that you must have already created in previous sections.

The reverse method returns a converter between the target system and the source. An implementation of this interface will typically use a multiplier to convert from the base unit of the source system to the destination. For example, to convert from the international system to the imperial, we will use the fact that a meter is equivalent to 3,280839895 feet. The converter returned by the reverse method will simply use the inverse (reciprocal) of this multiplier.

To improve the usability of the library, each metric system will have a *registry*, where the converters can be added. In this way, you must extend the IMetricSystem interface with a method to obtain the converter to a specific metric system.

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

As you see in the following example, by default it is not possible to convert from kilometers to miles. Once a converter is registered in SiLengthMetricSystem, then it must be possible to perform the conversion. When registering a converter, its inverse is registered as well. Don't forget to add the new exceptions that are necessary.

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

	IMagnitude inMiles = null;

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

	System.out.println("In miles = "+inMiles);
	System.out.println("In m = "+ inMiles.transformTo(SiLengthMetricSystem.METER));
  }
}
```

Expected output:

```
Cannot transform km L to ml L
In miles = 6.213711922348486 [ml L]
In m = 10000.0 [m L]
```

(*) Remark: You may get slightly different values due to rounding

Section 5: Class diagram and explanation of the design (0,5 points)

a) Create a class diagram showing the design you made. Do not forget that the purpose of the diagram is to explain the design with an adequate level of abstraction, it is not "Java in pictures". In this way, you must omit constructors, getters and setters, and you must represent the collections, arrays and references used as associations of the most appropriate type. Don't forget to include the interfaces you have used, and the implementation relationships between classes and interfaces. Include a small explanation of the design, as well as the decisions you have made.

b) Is your design extensible? Explain what steps should be given to:

b.1) add new units to an existing system

b.2) add a mass quantity to the international metric system

c) What disadvantages or limitations has the library?

Section 6 (Optional): Composite Units (1 point)

In this section you need to implement composite units, like m/s. For this purpose, one option is to extend the IPhysicalUnit interface as follows:

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

Where Operator is an enumeration or a class that represents a binary arithmetic operator (for simplicity, consider only * and /), getLeftUnit and getRightUnit return the left and right units (that can be *simple* or *composite*). A composite unit can be converted to another one (with method canTransformTo) if the left and right units can be transformed. To convert a magnitude d between composite units, d is multiplied with the result of operating (with * or /, depending on the operator of the composite unit) the results of converting 1.0 to the left and right units. That is, if we want to convert 10 m/s to ml/h (miles/hour), we have to:

1.0 m = 0.000621371 ml, and 1.0 s = 0.000277778 h, so that 10 m/s = 10 * 0.000621371/0.000277778 = 22,3693381 ml/h.

Take into account that composite units imply implementing a composite Quantity.

As an example, the following program should result in the output below:

```
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));       // implies converting m/s to miles/hour
	System.out.println(velocSI2.add(velocSI));       // implies converting m/s to km/hout
  }
}
```

Expected output

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

How to submit:

You should submit

- an **src** folder with the Java code of all sections, including test data, and additional tester that you have developed
- a **doc** folder with the generated documentation
- a PDF file with section 5
- If you have done the optional section, submit it in another project or folder

You have to pack everything in a unique ZIP file, named as follows: GR<groupnumber>_<studentnames>.zip. For example, for Marisa and Pedro of group 2213, the file name is: GR2213_MarisaPedro.zip.