Entregable Teórico 1

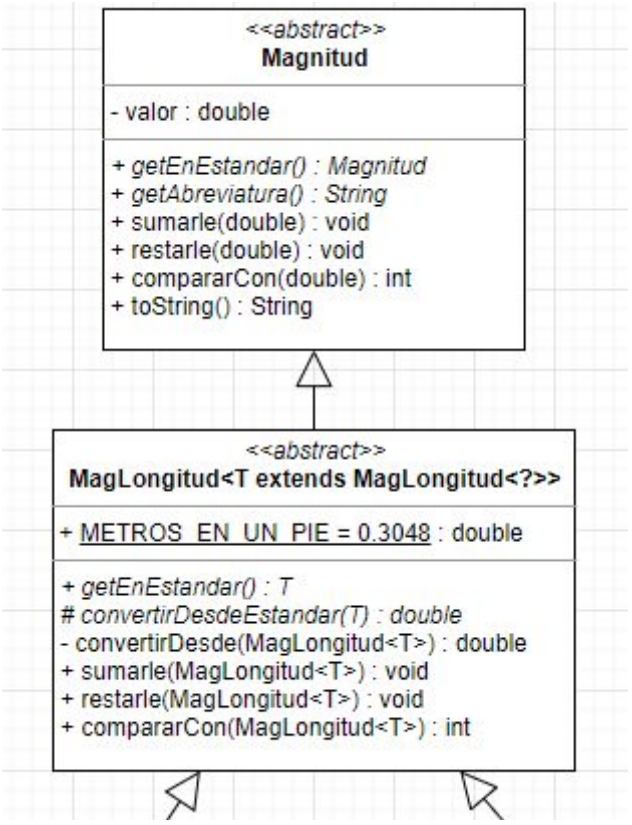
Alumnos: Calderón, Sergio Leandro
Ercoli, Juan Martín

Fecha de entrega: 24 de noviembre de 2020

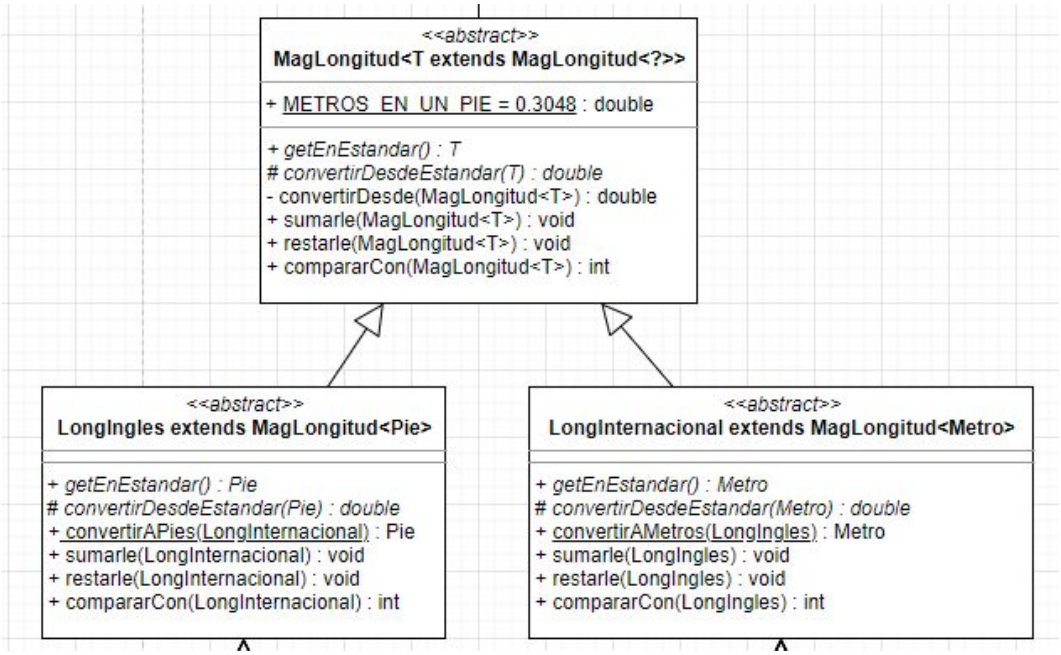
1) **¿Cómo han representado en su proyecto las magnitudes de longitud del sistema internacional y el sistema inglés?, ¿cuántas instancias posibles tiene cada una?, ¿de qué otras maneras es posible representar estas magnitudes?. Describan su solución y las decisiones tomadas.**

   Cada una de las magnitudes fue representada mediante clases concretas, que son subclases de clases abstractas que representan el tipo de magnitud (longitud, masa, tiempo) en un sistema determinado (inglés o internacional).

   Todas las magnitudes heredan el atributo **valor** de la clase **Magnitud**, y deben implementar cómo convertirse al estándar de su sistema, además de devolver su abreviatura específica (es una constante privada en cada una). Respecto al **toString()**, que lo utiliza la Sonda Espacial (ver puntos 5 y 6), está implementado en **Magnitud** porque simplemente consiste en devolver un String con el valor (con 4 decimales) seguido de la abreviatura.



Se tienen 3 tipos de magnitudes: **MagLongitud, MagTiempo** y **MagMasa**, en las cuales se especifica la unidad estándar en el tipo T. Estas clases definen un método abstracto **convertirDesdeEstandar** (inversa de **getEnEstandar**).



Se tienen 2 sistemas de longitud: **LongIngles**, que define a Pie como su unidad estándar; y **LongInternacional**, que define a Metro como su estándar. En estas dos clases se encuentran los métodos conversores entre sistemas.

Respecto a las clases concretas, se tienen las clases: Pie, Pulgada, Yarda y Milla, cuya superclase es **LongIngles** (ver UML en la página siguiente).

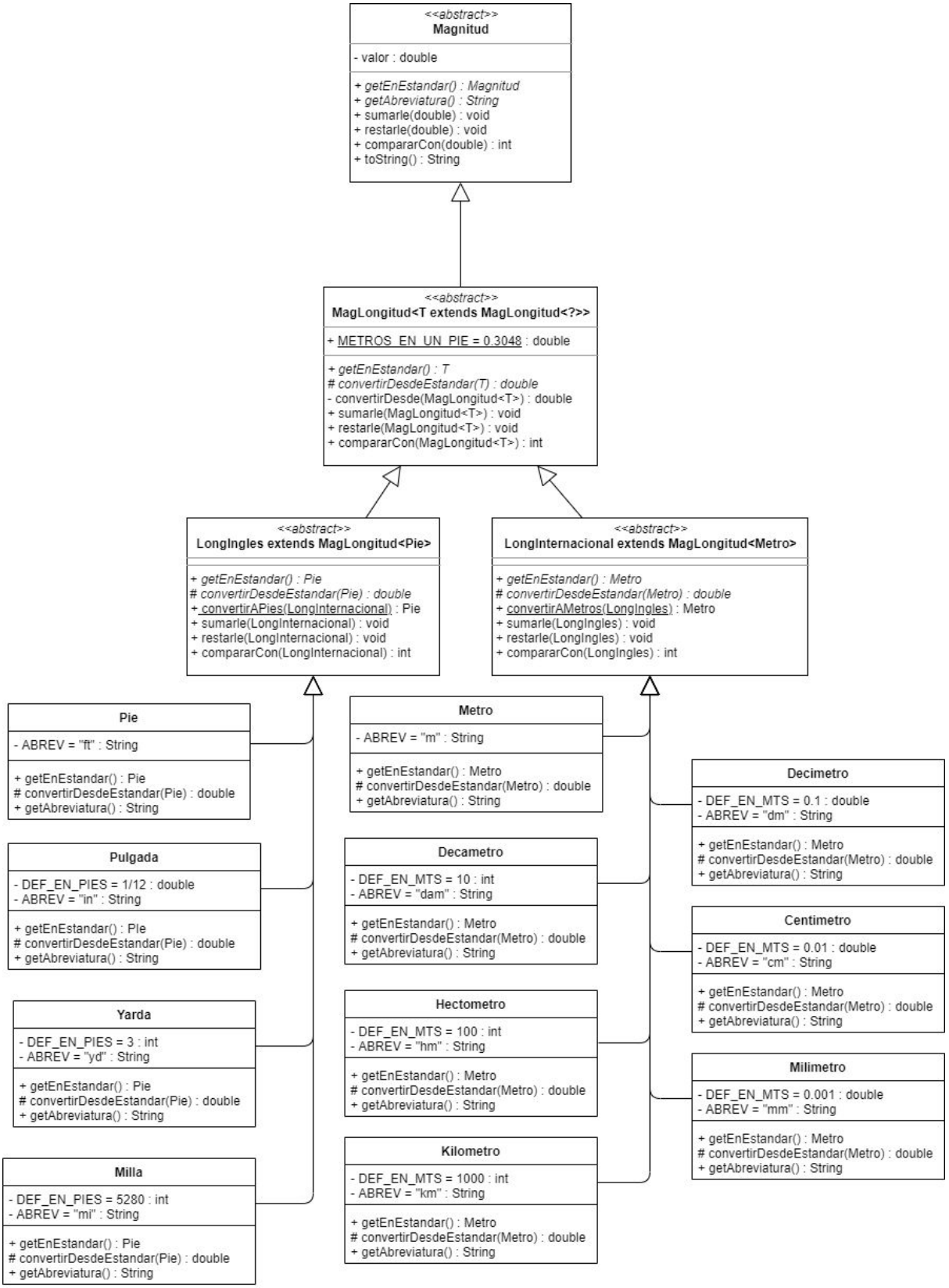
Por otro lado, se tienen las clases: Metro, Decímetro, Centímetro, Milímetro, Decámetro, Hectómetro y Kilómetro, cuya superclase es **LongInternacional.**

En nuestro proyecto, como ninguna de las clases es Singleton, se pueden instanciar tantas magnitudes como uno desee. Los constructores son públicos y requieren un valor inicial como parámetro.

Otra manera posible de representar las magnitudes es mediante el uso de **enumerativos**, en los cuales se debería especificar la equivalencia con el estándar de su sistema (un float) y la abreviatura (un String). Ejemplo:

```
public enum LongitudesInternacional {
    DECIMETRO( equivalencia: 0.1,  abreviatura: "dm"), CENTIMETRO( equivalencia: 0.01,  abreviatura: "cm"),
    METRO( equivalencia: 1,  abreviatura: "m"),  KILOMETRO( equivalencia: 1000,  abreviatura: "km");
```

UML de las magnitudes de longitud en nuestro proyecto:



2) **Identifiquen el código de su proyecto donde usaron interfaces, ¿cuál fue el motivo de su selección?, ¿podrían haber usado clases abstractas?, ¿cuál sería la diferencia en su proyecto entre usar interfaces o clases abstractas?**

En nuestro proyecto, no se utilizaron interfaces, pero sí clases abstractas con métodos abstractos. Se podría haber reemplazado la clase abstracta **Magnitud** por una interface, pero se decidió dejarla como tal porque la misma tiene un atributo **valor** de tipo **double** (una interfaz no almacena variables).

```java
public abstract class Magnitud {
    private double valor;

    public Magnitud(double valor) { this.valor = valor; }

    /**
     * Convierte la magnitud actual en el estandar de su sistema
     * @return una instancia de la magnitud estandar con el valor convertido
     */
    public abstract Magnitud getEnEstandar();

    /**
     * Permite obtener la abreviatura de la unidad en que se está trabajando
     * @return un cadena de texto con pocos caracteres (abreviatura de la unidad)
     */
    public abstract String getAbreviatura();
```

A su vez, además de obtener y setear dicho valor, la clase Magnitud también tiene métodos concretos para **sumar, restar y comparar** con **double**, lo cual es útil si se está trabajando con valores de una misma magnitud concreta.

Encontramos también que el método **compareTo**(Magnitud) de la interfaz Comparable traería un problema: downcasting, porque la clase Magnitud desconoce con qué clase concreta se está trabajando, y además estaría permitiendo comparar Longitud con Tiempo o Masa.

Respecto a la diferencia entre usar interfaces o clases abstractas: Si hubiéramos usado una interfaz en lugar de clase abstracta, tendríamos el problema mencionado anteriormente (las interfaces no almacenan variables), entonces se tendría que colocar el atributo **valor** en las subclases directas de Magnitud, es decir, en **MagLongitud, MagTiempo y MagMasa.** Además, para que la Sonda Espacial pueda acceder al mismo usando Magnitud, estas clases deberían implementar un **getValor()** que sería abstracto en Magnitud.

3) **De acuerdo a la especificación del proyecto se espera que puedan compararse unidades de longitud diferentes: ¿cómo han codificado esta funcionalidad?**

Para comparar unidades de longitud existen 3 métodos, de igual nombre, pero que se diferencian en el tipo de argumento que reciben.

Si se necesita comparar entre unidades del mismo sistema, por ejemplo, centímetro y decímetro, se utiliza el método definido en **MagLongitud:**

```
public int compararCon(MagLongitud<T> otroMismoSist){
    return super.compararCon(this.convertirDesde(otroMismoSist));
}

private double convertirDesde(MagLongitud<T> otroMismoSist){
    return this.convertirDesdeEstandar(otroMismoSist.getEnEstandar());
}

@Override
public abstract T getEnEstandar();

protected abstract double convertirDesdeEstandar(T estandar);
```

Cabe destacar que todas las unidades concretas saben convertirse a su estándar, y viceversa; es decir, todas las unidades de longitud del Sistema Internacional saben convertirse a Metros, y a su vez, pueden convertir un valor en Metros a su unidad. Lo mismo ocurre en el Sistema Inglés con Pies.

En nuestro proyecto, se decidió convertir siempre el argumento a la unidad con la cual se está trabajando (la instancia). Siguiendo con el ejemplo, los 5 decímetros se convertirán en 0.5 metros, y luego se convertirán en 50 cm.

Esto es posible gracias a que todas las unidades concretas tienen implementado los métodos **getEnEstandar** y **convertirDesdeEstandar**.

```java
public class Decimetro extends LongInternacional {
    private static final String ABREV = "dm";
    private static final double DEF_EN_MTS = 0.1;

    @Override
    public Metro getEnEstandar() {
        return new Metro( valor: super.getValor() * DEF_EN_MTS);
    }

    @Override
    protected double convertirDesdeEstandar(Metro estandar) {
        return estandar.getValor() / DEF_EN_MTS;
    }
```

Una vez que el argumento esté en la misma unidad que la instancia, se procede a llamar al método **compararCon** de la superclase **Magnitud**:

```java
/**
 * Compara el valor actual de la magnitud con otro de la misma magnitud
 * @param otroValor el otro valor a comparar
 * @return -1 si el actual es menor, 0 si son iguales, 1 si el actual es mayor
 */
public int compararCon(double otroValor){
    return Double.compare(this.getValor(), otroValor);
}
```

En resumen, los pasos a seguir, en un mismo sistema, serían los siguientes:

1. Convertir el argumento a su estándar (Metros o Pies)
2. Convertir el argumento a la unidad de la instancia de trabajo
3. Comparar los valores *doubles* de ambas unidades en la superclase

Por otro lado, para comparar unidades de longitud de diferentes sistemas, sólo se agrega un paso adicional de conversión, resultando la secuencia:

1. Convertir el argumento a su estándar (Metros o Pies)
2. Realizar la conversión a mi sistema (Metros a Pies, o viceversa)
3. Convertir el argumento a la unidad de la instancia de trabajo
4. Comparar los valores *doubles* de ambas unidades en la superclase

A modo de ejemplo, se muestra la comparación entre Centímetro y Pulgadas:

```
public void testComparacion(){
    Centimetro mCenti = new Centimetro( valor: 20);
    Pulgada mPulgada = new Pulgada( valor: 36);
    mCenti.compararCon(mPulgada);
}
```

Clase LongInternacional: las pulgadas se convierten en Pies y luego a Metros

```
public int compararCon(LongIngles longIngles){
    return super.compararCon(convertirAMetros(longIngles));
}

public static Metro convertirAMetros(LongIngles longIngles){
    return new Metro( valor: longIngles.getEnEstandar().getValor() * METROS_EN_UN_PIE);
}
```

```
tp02.magnitudes.longitud.MagLongitud<T extends MagLongitud<?>>
public static final double METROS_EN_UN_PIE = 0.3048
```

Clase Pulgada: la conversión a su estándar (Pie) consiste en dividir por 12.

```
public class Pulgada extends LongIngles {
    private static final String ABREV = "in";
    private static final double DEF_EN_PIES = 1d/12;

    @Override
    public Pie getEnEstandar() {
        return new Pie( valor: super.getValor() * DEF_EN_PIES);
    }
```

La referencia **super** del método **compararCon** es a la clase **MagLongitud,** es decir, la comparación entre unidades de un mismo sistema, lo cual ya se explicó anteriormente. Para este caso, las 36 pulgadas, que se convirtieron en 3 pies y luego en 0,9144 metros, proceden a convertirse en 91,44 cm.

El último paso se realiza en **comparar(double)** de la clase Magnitud, donde directamente se compara el atributo valor = 20 de la instancia de Centímetro, con el valor 91,44 que representan las 36 pulgadas en centímetros. En este caso, el método retorna el valor -1, porque 20 es menor a 91,44.

---

4) **Se espera de la solución que sea posible sumar y restar entre unidades de la misma magnitud pero de diferente sistema métrico particularmente entre el inglés y el internacional: ¿cómo han implementado este requerimiento?**

Definimos 2 métodos de instancia en la clase abstracta **Magnitud**:

- **sumarle(double valor):** le suma al atributo valor de la magnitud, otro valor que se encuentra en la misma magnitud.
- **restarle(double valor):** le suma al atributo valor de la magnitud, un valor que se encuentra en la misma magnitud.

En la clase abstracta **MagLongitud** definimos 2 métodos de instancia:

- **sumarle(MagLongitud<T>):** convierte la magnitud de longitud pasada como parámetro, que es del mismo sistema, a la unidad de longitud de la instancia que invoca al método y llama al **sumarle** de la superclase Magnitud con la magnitud convertida en la nueva unidad.
- **restarle(MagLongitud<T>):** convierte la magnitud de longitud pasada como parámetro, que es del mismo sistema, a la unidad de longitud de la instancia que invoca al método y llama al **restarle** de la superclase Magnitud con la magnitud convertida en la nueva unidad.

En cada **clase de sistema de longitud** definimos 1 método estático:

- **convertirAPies / convertirAMetros:** se recibe, como parámetro, una instancia de **LongInglesa** ó **LongInternacional** (según corresponda) y lo convierte en la unidad estándar del sistema opuesto.

También en cada **clase de sistema de longitud** se definen 2 métodos de instancia referidos a la suma y resta en distintos sistemas:

- **sumarle(LongInglesa ó LongInternacional):** primero **convierte el argumento a la unidad estándar del sistema de la instancia que invoca el método,** y luego, se suman los valores correspondientes en el método **sumarle** de la superclase MagLongitud.
- **restarle(LongInglesa ó LongInternacional):** primero **convierte el argumento a la unidad estándar del sistema de la instancia que invoca el método,** y luego, se restan los valores correspondientes en el método **restarle** de la superclase MagLongitud.

La implementación de estos métodos se puede ver de forma detallada en las siguientes imágenes, mostrando también los demás métodos invocados:

Clase Magnitud:

```java
/**
 * Actualiza el valor de la magnitud, sumándole un nuevo valor en la misma magnitud
 * @param valor el nuevo valor a añadir sobre el actual
 */
public void sumarle(double valor) { this.valor += valor; }

/**
 * Actualiza el valor de la magnitud, restándole un nuevo valor de la misma magnitud
 * @param valor el nuevo valor a sustraer del actual
 */
public void restarle(double valor) { this.valor -= valor; }
```

Clase MagLongitud<T>:

```java
public void sumarle(MagLongitud<T> otroMismoSist){
    // 1. Convierte el argumento a la unidad que estoy trabajando, pasando por el estándar
    // 2. Adiciona el valor en double en la superclase (ya están en la misma unidad)
    super.sumarle(this.convertirDesde(otroMismoSist));
}
```

```java
public void restarle(MagLongitud<T> otroMismoSist){
    // 1. Convierte el argumento a la unidad que estoy trabajando, pasando por el estándar
    // 2. Sustrae el valor en double en la superclase (ya están en la misma unidad)
    super.restarle(this.convertirDesde(otroMismoSist));
}
```

```java
/**
 * Convierte desde una magnitud del mismo sistema hacia la magnitud actual
 * @param otroMismoSist la magnitud que nos interesa convertir
 * @return el valor de la magnitud en la unidad actual
 */
private double convertirDesde(MagLongitud<T> otroMismoSist){
    // 1. Convierte el argumento al estándar (Metros o Pie según sistema)
    // 2. Convierte desde el estándar a la unidad de mi instancia (Ejemplo: m a cm)
    return this.convertirDesdeEstandar(otroMismoSist.getEnEstandar());
}
```

```java
/**
 * Convierte desde la magnitud estándar hacia la magnitud actual
 * @param estandar la magnitud expresada en la unidad estándar (tipo genérico)
 * @return el valor de la magnitud en la unidad actual
 */
protected abstract double convertirDesdeEstandar(T estandar);
```

Clase LongInternacional:

```java
public void sumarle(LongIngles longIngles){
    // 1. Convierte cualquier magnitud del Inglés a Metro, pasando por Pie
    // 2. Adiciona en la superclase, ya que ahora son del mismo sistema
    super.sumarle(convertirAMetros(longIngles));
}
```

```java
public void restarle(LongIngles longIngles){
    // 1. Convierte cualquier magnitud del Inglés a Metro, pasando por Pie
    // 2. Sustrae en la superclase, ya que ahora son del mismo sistema
    super.restarle(convertirAMetros(longIngles));
}
```

```java
public static Metro convertirAMetros(LongIngles longIngles){
    // 1. Convierte la magnitud a Pies (si ya son Pies no lo hace)
    // 2. Convierte de Pies a Metros mediante la única equivalencia entre sistemas
    return new Metro( valor: longIngles.getEnEstandar().getValor() * METROS_EN_UN_PIE);
}
```

Clase LongInglesa:

```java
public void sumarle(LongInternacional longDelSI){
    // 1. Convierte cualquier magnitud del S.I a Pie, pasando por Metro
    // 2. Adiciona en la superclase, ya que ahora son del mismo sistema
    super.sumarle(convertirAPies(longDelSI));
}
```

```java
public void restarle(LongInternacional longDelSI){
    // 1. Convierte cualquier magnitud del S.I a Pie, pasando por Metro
    // 2. Sustrae en la superclase, ya que ahora son del mismo sistema
    super.restarle(convertirAPies(longDelSI));
}
```

```java
public static Pie convertirAPies(LongInternacional longDelSI){
    // 1. Convierte la magnitud a Metros (si ya son Metros no lo hace)
    // 2. Convierte de Metros a Pies mediante la única equivalencia entre sistemas
    return new Pie( valor: longDelSI.getEnEstandar().getValor() / METROS_EN_UN_PIE);
}
```

La secuencia de pasos para las operaciones de suma y resta entre diferentes unidades es muy similar a la que se realiza para la comparación (punto 3).

A modo de ejemplo, se muestra la suma entre decametros y yardas:

```
Decametro decametro = new Decametro( valor: 20);
Yarda yarda = new Yarda( valor: 15.2);
decametro.sumarle(yarda);
```

```
20.0 dam + 15.2 yd = 21.3899 dam
```

- ➢ Primero se llama a **sumarle(LongIngles)** donde las yardas se convierten en metros (pasando por el estándar inglés: pies) y luego se llama al método de instancia **sumarle** de la clase padre MagLongitud.
- ➢ En el **sumarle de MagLongitud** se convierten los metros (que antes eran yardas) a la unidad en la que estamos trabajando (Decámetro) y luego se llama al método **sumarle** de la clase padre Magnitud.
- ➢ En **sumarle(double)** de Magnitud se suman los valores directamente, ya que las yardas ya fueron convertidas a decámetros.

---

5) De acuerdo al enunciado, la clase SondaEspacial es un tipo genérico, ¿cuáles serían ejemplos de tipos parametrizados que se pueden obtener? ¿se trata de un tipo genérico con alguna cota? Si observan alguna ventaja definiendo a SondaEspacial como un tipo genérico, describanla.

La clase SondaEspacial está declarada como una clase concreta que recibe un tipo genérico **T extends Magnitud**, por lo tanto, admite todas las magnitudes de longitud, masa y tiempo, y de cualquier sistema. Ejemplos:

```
SondaEspacial<Magnitud> sonda1 = new SondaEspacial<>();
SondaEspacial<Metro> sonda2 = new SondaEspacial<>();
SondaEspacial<LongIngles> sondaInglesa = new SondaEspacial<>();
SondaEspacial<MagLongitud<Centimetro>> sondaCGS = new SondaEspacial<>();
```

Como se mencionó anteriormente, la cota es que el tipo genérico sea de tipo **Magnitud** o alguna de sus subclases, abstractas o concretas. La ventaja de definirlo de esta manera es que se puede asegurar que la sonda opere con instancias para uso específico. Por ejemplo, la sonda no admitirá String.

Además, como la sonda permite imprimir un argumento tipo T, también se asegura que, además del *toString()*, se puedan utilizar los métodos definidos en la clase **Magnitud**, tales como *sumar(double)* y *getEnEstandar()*. En caso de utilizar estos últimos, se evita tener que realizar un *downcasting*.

6) **¿Cuál es la interface pública y cuál la privada de la clase SondaEspacial? ¿Cuáles fueron los criterios de decisión de ubicar determinados métodos en una o en otra?**

La interface pública está compuesta por un método de instancia llamado **imprimirArgumento(T arg),** que recibe un argumento de tipo T (T extends Magnitud), y por el **constructor por defecto**.

Método imprimirArgumento(T arg):

```java
/**
 * Muestra los detalles del argumento pasado: su valor, abreviatura y conversión en el estándar
 * @param arg la instancia de magnitud que se desea conocer su información
 */
public void imprimirArgumento(T arg){
    System.out.println("<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>");
    System.out.println("Recibí: " + arg);
    System.out.println("Su valor exacto es: " + arg.getValor());
    System.out.println("En su estándar, es: " + arg.getEnEstandar());
    System.out.println("<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>\n");
}
```

La clase SondaEspacial no tiene atributos ni métodos privados, el constructor es público, no se trata de un Singleton, y además, los métodos de Magnitud que invoca en imprimirArgumento son públicos para todas las clases, por lo tanto, decimos que SondaEspacial no tiene una interface privada relevante.

Ubicamos a imprimirArgumento en la interface pública ya que entendimos que este método puede ser utilizado en un contexto externo a la clase.

Respecto a las conversiones: son totalmente ajenas a la SondaEspacial y ésta no tiene asignada la labor de convertir magnitudes, por eso mismo decidimos hacerlas en clases fuera de la SondaEspacial. Por ejemplo, si la SondaEspacial fue instanciada de forma que sólo acepte LongInternacional, se tendrá que realizar una conversión externa para LongInglés:

```java
SondaEspacial<LongInternacional> sondaEspacial = new SondaEspacial<>();

for (LongIngles magIngles : otrasMagnitudes) {
    System.out.println("Iterando sobre: " + magIngles);
    Metro enMetros = LongInternacional.convertirAMetros(magIngles);
    sondaEspacial.imprimirArgumento(enMetros);
}
```