Junco de las Heras & Marta Vaquerizo

PRÁCTICA 4
ANÁLISIS Y DISEÑO DE SOFTWARE

Apartado 1:

Se implementa la interfaz **IPhysicalUnit** y la clase **Quantity** que sirve para guardar la información de qué tipo de medida es cada objeto. Se crea la excepción **QuantityException**, que se lanza cuando se intenta transformar unidades en las que la **Quantity** difiere. También se crean las clases **SiLengthMetricSystem** y **SiTimeMetricSystem**, y algunas unidades como pueden ser el **Meter**, **Kilometer**, **Second**… Estas clases solo serán accesibles a través de su clase padre a través de una variable estática final. Su clase padre, a la vez, presenta un patrón de diseño de Singleton, donde solo es posible crear una instancia de ella.

Apartado 2:

Se implementa la interfaz **IMetricSystem** y **MetricSystem** y 3 unidades para cada sistema: Sistema Si de tiempo: **Hour**, **Milisecond** y **Second**. Sistema si de longitud: **Kilometer**, **Meter** y **Milimeter**. Sistema Imperial de longitud: **Foot**, **Mile**, **Yard**. **IPhysicalUnit** e **IMetricSystem**, implementan las funciones **base ()**, que devuelve la unidad fundamental del sistema, y **units ()**, que es una Colección de las unidades que hemos implementado de ese sistema.

Apartado 3:

En este apartado, se ha implementado la interfaz **IMagnitude** y la clase **Magnitude**, que la implementa. Esta clase tiene dos atributos que la definen: un valor, de tipo double, y una unidad, de tipo **IPhysicalUnit**. Además, se ha descomentado la función **IMetricSystem getSystem ()** de la interfaz **IPhysicalUnit**, para poder manejar el sistema métrico al que pertenecen las unidades. Esta función se ha implementado en todas las clases que implementan la interfaz **IPhysicalUnit**.

Como hemos comentado anteriormente, esta clase implementa la interfaz **IMagnitude**, y por ello, implementa las funciones de la misma. Se van a comentar las que conllevan más implementación.

Las funciones **IMagnitude add (IMagnitude m)**, **IMagnitude subs (IMagnitude m)** e **IMagnitude transformTo (IPhysicalUnit c)** comprueban que la magnitud que se le pasa como argumento (m2) es compatible con la que llama a la función (m1), y en caso de que no, lanzar **QuantityException**. La función **add** crea una magnitud cuyo valor es la suma de los valores de ambas magnitudes (m1 y m2) en la unidad de la magnitud m1. La función **subs** crea una magnitud cuyo valor es la resta entre el valor de m1 y el valor de m1 (m1.valor – m2.**getValor ()**), en la unidad de m1. Y la función **transformTo**, crea una magnitud cuyo valor es el valor de m1 pasado a la unidad u1, pasada por argumento, siempre que el sistema métrico de la u1 es el mismo que el de la unidad de m1.

Tras implementar la clase y la interfaz mencionadas, el **MainTest** funciona tal y como se esperaba.

Apartado 4:

En este apartado, se han implementado la interfaz **IMetricSystemConverter**, y las clases **AbstractMetricSystem**, clase abstracta con los elementos comunes a todos los sistemas métricos; **SiLength2ImperialConverter** e **Imperial2SiLengthConverter**, conversores entre el sistema métrico internacional y el imperial de longitud, que implementan la interfaz **IMetricSystemConverter**; y **UnknownUnitException**, que hereda de **QuantityException**.

Por otro lado, se ha añadido la función **IMetricSystemConverter getConverter (IMetricSystem to)** a la interfaz **IMetricSystem**, teniendo que ser implementada en todas las clases

que implementen la interfaz. Esta función te devuelve un conversor entre el sistema métrico que lo llama y el pasado por argumento "to".

En la clase **AbstractMetricSystem** se ha incluido, aparte del quantity, que compartían todos los sistemas métricos, y sus funciones asociadas (**setQuantity** o **getQuantity**), una colección con los conversores del sistema métrico, tanto en los que el sistema es fuente como en los que es destino.

En cuanto a las funciones de la interfaz **IMetricSystemConverter**, se van a comentar por encima. Por un lado, están los getters **IMetricSystem sourceSystem () e IMetricSystem targetSystem ()**, que te devuelven el Sistema métrico fuente y el destino, respectivamente. Y por otro lado están **IMagnitude transformTo (IMagnitude from, IPhysicalUnit to)** e **IMetricSystemConverter reverse ()**. Esta última, devuelve un conversor entre el sistema destino y fuente. Y la función transformTo, primero comprueba que la unidad destino pertenece al sistema destino. Si no pertenece, se lanza la excepción **UnknownUnitException**, y si pertenece, crea una magnitud, cuyo valor es el valor de la magnitud from en la unidad pasada por argumento to.

Tras estos cambios, el **ConversionTest** funciona como se esperaba.

Apartado 5:

a) El diagrama de clases es el fichero **diagrama_clases.png**. También está al final de este pdf. Está formado por packages, que tienen clases y relaciones entre clases. También está el package de las exceptions, pero no tienen relaciones con los otros packages debido a que son las funciones las que lanzan las excepciones y no las clases. Los packages de si.length, si.time e imperial.length tienen una estructura similar, una clase general que los describe y 3 unidades implementadas de cada sistema. Estas clases heredan de **AbstractMetricSystem**, que a su vez implementa **IMetricSystem e IPhysicalUnit**. Luego está el package si.lengthConverters, que incluyen a los conversores, que implementan **IMetricSystemConverters**. También está el package magnitude, de las magnitudes, que tienen una asociación con las **Quantity**, que están en el package units.

En el diagrama de clases se pueden ver bien las relaciones de herencia y las de pertenencia, así como la navegabilidad. En cada clase están los métodos característicos de ella, así como de sus atributos.

b) Nuestro diseño ha intentado ser lo más extensible posible siguiendo los modelos del paradigma de Orientación a Objetos (OO), reutilizando las funciones ya implementadas.

b1) Para ello habría que crear una clase con el nombre de la unidad y extender a la clase del sistema que pertenece (por ejemplo **SiLengthMetricSystem**), implementando el constructor, **abbrev** y **toString**, y luego habría que añadir esta unidad como una variable dentro de **SiLengthMetricSystem**. Esto es así porque es un requisito que desde afuera solo se pueda crear 1 clase cada tipo, así que la variable será final y hay que añadirla cada vez que se crea una unidad nueva.

b2) Se crearía la clase **SiMassMetricSystem**, que hereda de **AbstractMetricSystem** y se tendrían que implementar los métodos de las interfaces **IPhysicalUnit** e **IMetricSystem**, que son **canTransformTo**, **transformTo**, **base**, **units**, **abbrev**, **getMetricSystem**. Si se quiere utilizar los conversores también deben de implementarse **registerConverter** y **getConverter**.

c) Una de las limitaciones que tiene este modelo que hemos creado es que no hay una automatización para añadir unidades ni sistemas nuevos. Esto no lo hemos necesitado, pero podría ser conveniente si se necesitase, es decir, un método que crease nuevas unidades. Como cada nueva unidad es representada por una clase, esto hace que la creación de nuevas unidades un tanto tediosa.

Otro defecto de este diseño es que no es dinámico, es decir, que en nuestro modelo 1 metro siempre va a ser 3.280839895 pies, pero hay algunas unidades que fluctúan ligeramente con el tiempo.

Apartado 6:

Se crea la interfaz **ICompositeUnit**, donde se guardan tres cosas, dos magnitudes (l, y r) y un **Operador** (o). Se ha implementado el operador / y el operador *, así que puede representarse una Unidad Compuesta de forma recursiva, como l/r, donde l y r pueden ser unidades (Compuestas o no). Se ha creado la clase **CompositeUnit**, que implementaba la interfaz, para poder probar el test **CompositeTest**, funcionando como lo esperado.

Diagrama de clases:
(Es una miniatura, para poder ampliar ver **diagrama_clases.png**)