**Definiciones:**

**Ingeniería de software:** es la aplicación de un proceso sistemático, disciplinado y cuantificable al proceso de desarrollo, operación y mantenimiento de software.

**Modelo:** una representación o especificación desde un determinado punto de vista y con un objetivo concreto.

**Diseño:** conjunto de planes y decisiones para definir un producto con suficiente con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos.

**Patrón de diseño:** solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software.

**Diferencias patrón estado y patrón estrategia:**

El patrón estrategia es un patrón de diseño que se usa cuando se tienen distintos algoritmos para realizar la misma tarea y el cliente decide la implementación a usar en un momento dado. Se puede cambiar de implementación del algoritmo en tiempo de ejecución. Permite cambiar el algoritmo independientemente del cliente que los use. Por otro lado el patrón estado define una serie de comportamiento para un conjunto de operaciones de una clase que variarán dependiendo del estado en el que se encuentre nuestro objeto.

Ambos patrones se asemejan en que cambian en tiempo de ejecución el comportamiento o manera de resolver algo, sin embargo, uno se centra a nivel de método, el patrón estrategia, mientras que el patrón estado, se centra a nivel objeto.

**Tipo, Clase y DataType en UML. Semejanzas y diferencias:**

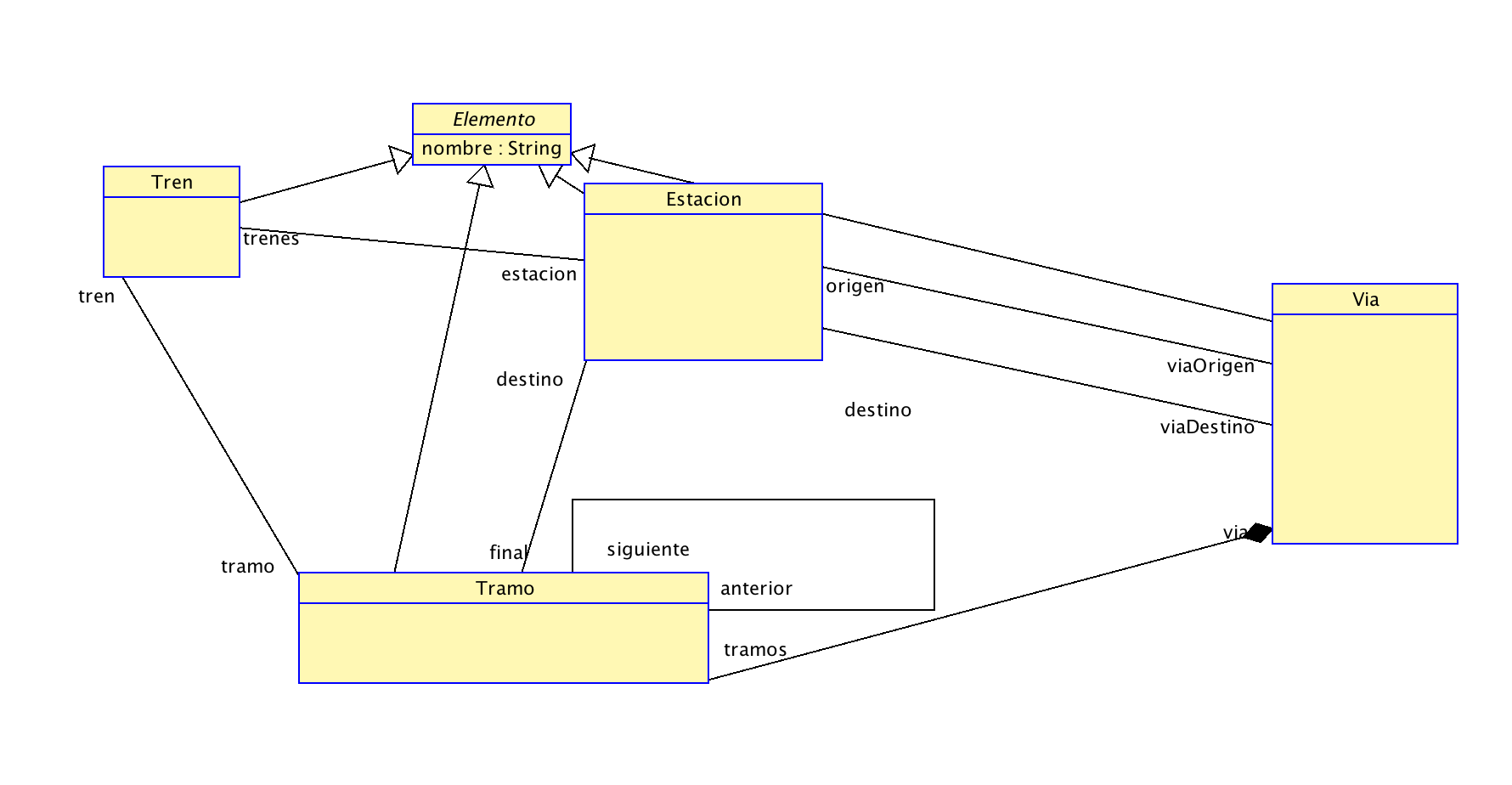
Un tipo es un predicado que caracteriza a un conjunto de elementos mientras que la clase es el conjunto de elementos que satisfacen ese tipo. Así pues, todos los elementos del modelado conceptual tienen un predicado que les caracteriza, mientras que una clase, es el conjunto de elementos de un tipo concreto.

El datatype no es más que un nuevo predicado que no estaba definido en el estándar, de este modo, un elemento puede cumplir un predicado más específico, que es el definido en el DataType.

**Ejercicio 1:**

1. En este ejercicio podemos observar la existencia de 4 clases principales distintas, que son Estación, Vía, Tramo y Tren. Además, se nos dice que todos los elementos existentes tienen una cadena de caracteres que los identificará, es decir, un nombre. Para no poner un atributo igual en todos los elementos del sistema, vamos a crear una clase abstracta Elemento con su atributo nombre, y que el resto de clases hereden de ella, para facilitar la implementación. En cuanto a las relaciones entre clases, una vía no es más que una relación reflexiva entre estaciones, una de origen y otra de destino. Cada vía se relaciona con un conjunto de tramos, que también tiene una relación reflexiva para saber qué tramo es el siguiente y cual es el anterior. Además, el último y el primer tramo estarán relacionados con la estación, ya que en el enunciado se nos pide específicamente que el primer tramo y el último estarán relacionados con las estaciones de origen y destino.

Diagrama UML en USE



1. Debido a como hemos decidido modelar nuestro sistema hay restricciones que vienen implícitas en las multiplicidades, como es la restricción de que en un tramo puede haber solo un único tren. El sistema cuenta con 3 restricciones más. Una en el contexto del tren, que te obliga a que un tren esté o en una estación, o circulando por un tramo, y en el contexto de tramo, que obliga a que un tramo tenga un siguiente y un anterior, o que esté en una estación, pero nunca ambos, o es siguiente, o es estación, al igual que o es anterior, o es estación. Por último, necesitaremos añadir una restricción que impida que un tramo esté conectado consigo mismo, evitando así ciclos en el modelo. Nuestro modelo quedaría complementada con las siguientes restricciones:

*constraints*

*context Tren*

*inv oViaOTramo:*

*(self.estacion->size() > 0 and self.tramo->size() = 0) or (self.estacion->size() = 0 and self.tramo->size() > 0)*

*context Tramo*

*inv todasViasConectadas:*

*((self.siguiente->size() > 0 and self.destino->size() = 0) or (self.siguiente->size() = 0 and self.destino->size() > 0))*

*or ((self.anterior->size() > 0 and self.destino->size() = 0) or (self.anterior->size() = 0 and self.destino->size() > 0))*

*inv noCiclos:*

*self.anterior <> self.siguiente*

1. Definiciones de las queries:

*Via::numTrenes(): Integer = self.tramos->iterate(t: Tramo; sum: Integer = 0 | sum + t.tren->size())*

*Via::numTramos(): Integer = self.tramos->size()*

*Tren::posicion(): String = if self.estacion->size() > 0 then self.estacion.nombre else self.tramo.nombre endif*

USE:

*model Trenes*

*abstract class Elemento*

*attributes*

*nombre: String*

*end*

*class Tren < Elemento*

*operations*

*posicion(): String = if self.estacion->size() > 0 then self.estacion.nombre else self.tramo.nombre endif*

*end*

*class Estacion < Elemento*

*end*

*class Via < Elemento*

*attributes*

*a: Integer*

*operations*

*-- marriages(): Integer = self.wife->union(self.husband)->size()*

*numTrenes(): Integer = self.tramos->iterate(t: Tramo; sum: Integer = 0 | sum + t.tren->size())*

*numTramos(): Integer = self.tramos->size()*

*end*

*class Tramo < Elemento*

*end*

*association estacionOrigen between*

*Estacion [1] role origen*

*Via [0..1] role viaOrigen*

*end*

*association estacionDestino between*

*Estacion [1] role destino*

*Via [0..1] role viaDestino*

*end*

*association estacionado between*

*Tren [\*] role trenes*

*Estacion [0..1] role estacion*

*end*

*association seEncuentra between*

*Tren [0..1] role tren*

*Tramo [0..1] role tramo*

*end*

*composition compuesto between*

*Via [1] role via*

*Tramo [1..\*] role tramos*

*end*

*association tramos between*

*Tramo [0..1] role anterior*

*Tramo [0..1] role siguiente*

*end*

*association tramoFinal between*

*Tramo [0..1] role final*

*Estacion [0..1] role destino*

*end*

*constraints*

*context Tren*

*inv oViaOTramo:*

*(self.estacion->size() > 0 and self.tramo->size() = 0) or (self.estacion->size() = 0 and self.tramo->size() > 0)*

*context Tramo*

*inv todasViasConectadas:*

*((self.siguiente->size() > 0 and self.destino->size() = 0) or (self.siguiente->size() = 0 and self.destino->size() > 0))*

*or ((self.anterior->size() > 0 and self.destino->size() = 0) or (self.anterior->size() = 0 and self.destino->size() > 0))*

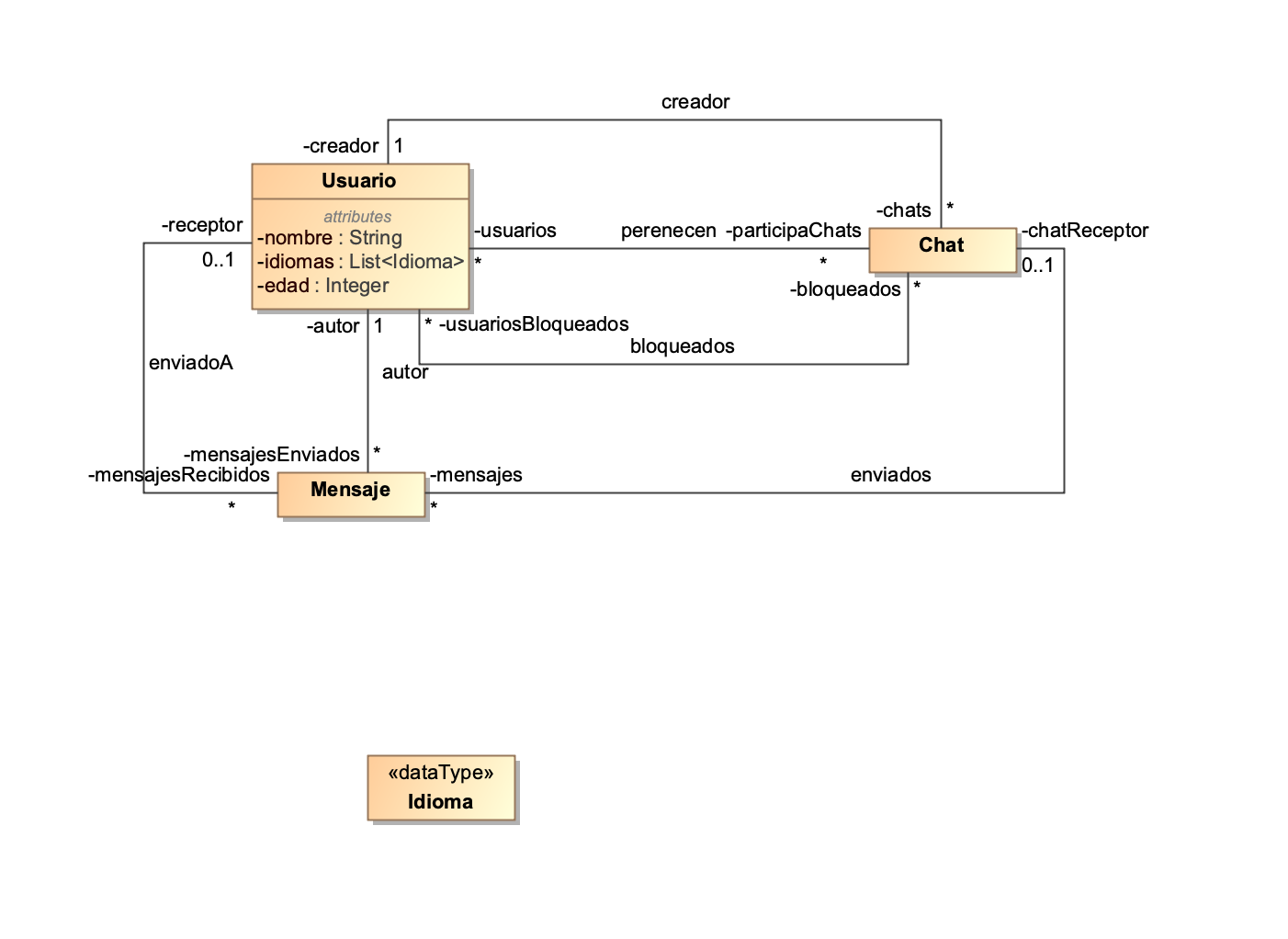
*inv noCiclos:*

*self.anterior <> self.siguiente*

**Ejercicio 2:**

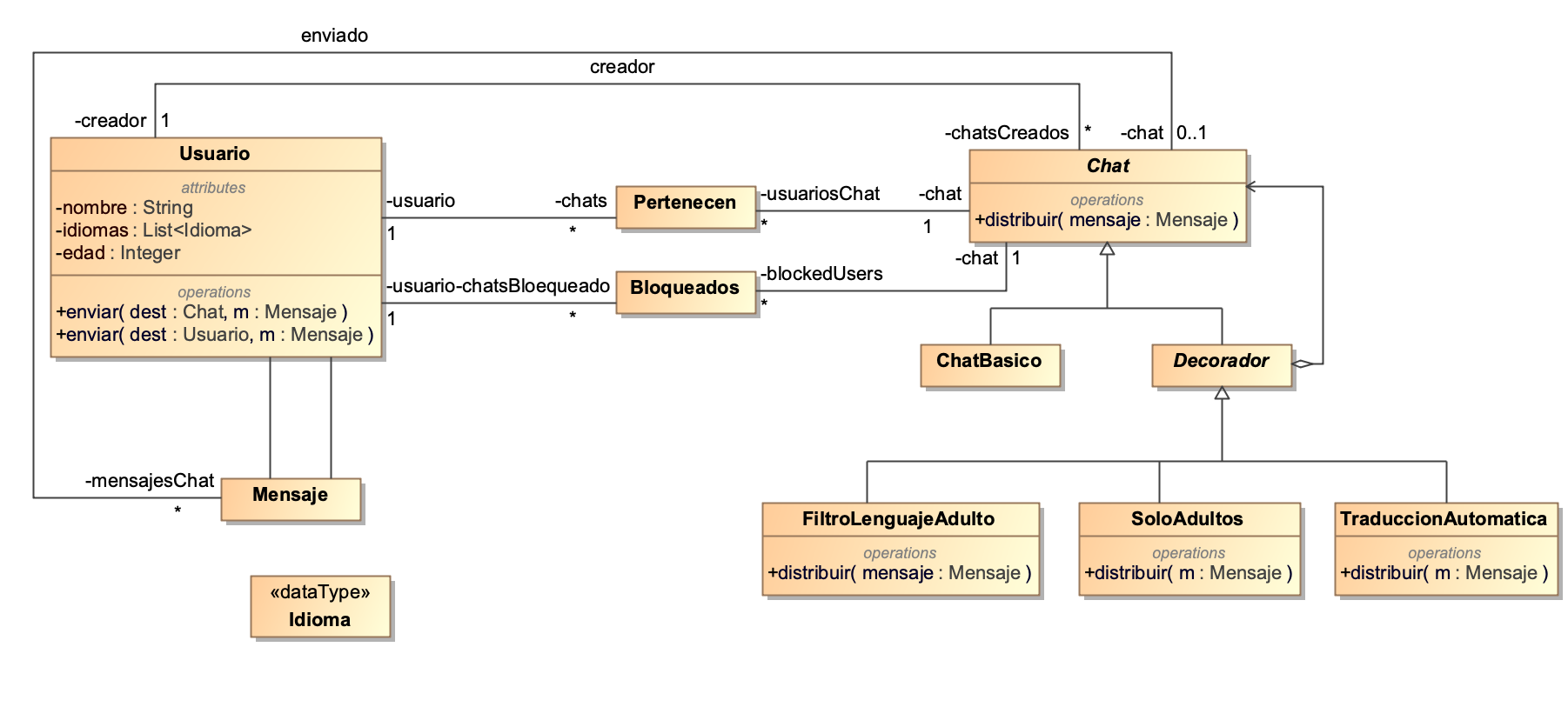
1. Nobody expected the Decorator pattern!!!!! \*pikachu sorprendido\*. Para resolver este problema vamos a definir 3 clases básicas, Usuario, Mensaje y Grupo. Los mensajes son enviados por un único usuario a otro usuario o a un chat. Los usuarios pertenecen a un chat, o pueden estar bloqueados. Además los chats son creados por un único usuario. El diagrama UML queda como sigue

UML



1. Para poder implementar esta solución en java, necesitaremos crear 2 clases de asociación sin atributos entre bloequeados y pertenecen para mantener la consistencia en el sistema. Además, los usuarios contarán con 2 operaciones similares, que le permiten mandar mensajes a un usuario o a un grupo. El grupo, al recibir un mensaje, lo redistribuye al resto de usuarios. Para poder implementar los filtros, usaremos el patrón decorador, para que cada chat tenga un seguido de características activadas y puedan combinarse entre ellos. Para mantener la consistencia necesitaremos añadir varias operaciones adicionales para insertar en listas y atributos de otros objetos los objetos correspondientes para tener el sistema bien montado.

Diagrama UML



1. Esquema de implementación en java que lo va a hacer peter el panda.

**Ejercicio 3:**

1. el sistema de ventanas lo tengo ya realizado en otro examen. Madre mía easy chaval.