Introducción a la Programación

Tema 3. Diseño de Tipos

1. Introducción

Un buen diseño de tipos es básico para que los programas sean comprensibles y fáciles de mantener. Veamos algunas pautas para este diseño y algunos ejemplos que puedan servir de guía.

Al diseñar un tipo nuevo debemos partir de los ya existentes. Es necesario decidir a qué otros tipos extender. En este capítulo vamos a ir viendo algunos tipos que podemos usar para

diseñar nuevos tipos. El nuevo tipo puede usar los tipos disponibles para declarar variables parámetros formales, etc. Decimos que el nuevo tipo **usa** esos tipos. También puede diseñarse el nuevo tipo extendiendo algunos de los disponibles. En este caso decimos que el nuevo tipo es un **subtipo** de los tipos de los que hereda o también decimos que **refina** esos tipos.

Todo tipo tiene, además de las heredadas de los tipos que refina, unas propiedades y posiblemente unas operaciones. Cada propiedad tiene un nombre, un tipo, puede ser **consultada** y además **modificada** o sólo consultada, y puede ser una **propiedad simple** o una **propiedad derivada**. Además las propiedades pueden ser **individuales** y **compartidas**. Cada tipo tiene una población. La **población de un tipo** es el conjunto de objetos que podemos crear de ese tipo. Como vimos en el capítulo 1 las propiedades individuales son específicas de un objeto individual. Las propiedades compartidas son comunes a todos los objetos de la población del tipo. Las propiedades derivadas pueden ser calculadas a partir de las otras propiedades. Las simples o básicas no. Las propiedades son usualmente consultables y pueden ser también modificables.

Las propiedades pueden tener parámetros y una **precondición**. Una precondición es una expresión lógica, construida a partir de los valores de del resto de las propiedades del tipo, que indica en qué condiciones es posible obtener el valor de la propiedad. Según sean modificables o sólo consultables, deduciremos un conjunto de métodos.

Justa a las propiedades, un tipo puede disponer de **operaciones**. Las operaciones son mecanismos para cambiar los valores de las propiedades del tipo. Una operación puede tener **parámetros**, **precondiciones** y **postcondiciones**. Precondiciones y postcondiciones son expresiones lógicas. Las primeras indican en qué condiciones es posible aplicar la operación. Las segundas indican relaciones que se deben cumplir entre los valores de alas propiedades antes y después de aplicar la operación. De las **operaciones** deduciremos otro conjunto de métodos.

Es importante al diseñar un tipo decidir si va a ser un **tipo mutable** o un **tipo inmutable**. Un tipo inmutable no tiene propiedades modificables y tampoco operaciones que puedan modificar los valores de las propiedades de un objeto. Si un tipo no es inmutable lo denominamos mutable.

Una cuestión importante al diseñar un tipo es indicar los **criterios de igualdad**. Una vez indicados estos criterios hay otras características del tipo que están completamente relacionadas y que estudiaremos a la vez: código hash, representación como cadena de caracteres de los objetos del tipo y un posible orden natural. Aquí es conveniente tener en cuenta los posibles órdenes que pueden ser interesantes para ordenar los objetos del tipo.

Además, para cada tipo individual o para varios tipos relacionados tenemos que diseñar una **factoría**. Una factoría, del tipo *T,* es un nuevo tipo cuyos métodos son diseñados para crear objetos del tipo *T*.

El conjunto de métodos deducidos de las propiedades y las operaciones del tipo forman los métodos del tipo. En Java los nuevos tipos se definen mediante un *enum*, un *interface* o una *clase*. El capítulo siguiente daremos algunas sugerencias para decidir si el tipo se concretará en un interface o en una clase (el caso del tipo *enum* es un caso particular).

El diseño del tipo, junto con los métodos que hemos escogido, debemos completarlo con ejemplos del funcionamiento de los mismos. Es decir debemos proporcionar como cambiarán los valores de las propiedades al aplicar determinadas operaciones. Es lo llamaremos **casos de prueba**. Estos casos de prueba, también denominados test de prueba, los completaremos junto con el diseño del tipo. En algunos casos será posible indicar restricciones entre los valores de las propiedades de un tipo. Estas restricciones las llamaremos **invariantes**. El conjunto de los métodos del tipo, sus casos de prueba, sus posibles invariantes, precondiciones y postcondiciones lo llamaremos **contrato del tipo**. El contrato de un tipo indica, por lo tanto, la forma de usar los objetos de un tipo (sus métodos) y lo que se espera de ellos.

Para diseñar un tipo podemos seguir la siguiente plantilla:

NombreDeTipo extiende T1, T2 …

- *El tipo es mutable o inmutable*
- *Propiedades*
    - *NombreDePropiedad*, Tipo, Consultable o no, derivada o no, compartida o individual. Un posible comentario.
    - …
- *Operaciones*
    - …
- *Definición de la Igualdad*
    - Criterio de Igualdad: detalles
    - Representación como cadena: detalles
    - Orden natural: si lo tiene especificar detalles
    - Otros órdenes posibles
    - …
- *Factoría*
    - Descripción de los métodos de la misma
- Casos de Prueba.
    - …

Ejemplo: el tipo Punto2D

Tipo Punto2D

- El tipo es mutable
- *Propiedades*
    - *X, Double, Consultable, Modificable, Individual, Básica*
    - *Y, Double, Consultable, Modificable, Individual, Básica*
    - *Origen, Punto2D, Consultable, Compartida*

    - *DistanciaA(Punto2D p), Double, Consultable, Individual.* Distancia a p.
    - *DistanciaAlOrigen, Double, Consultable, Individual.*
- *Operaciones*
    - *void mueveA(Double x, Double y).* Cambia las coordenadas a las x e y dadas.
- *Definición de la Igualdad*
    - Dos puntos son iguales si tienen iguales X e Y.
    - Un punto se representará en el formato (X,Y).
    - Su orden natural es: un punto es anterior a otro si tiene menor X, y si las tienen iguales, si tiene menor la Y.
    - Orden según las X
    - Orden según las Y
    - Orden según la distancia al origen
- *Factoría*
    - *Punto2D create(Double x, Double y).* Devuelve con las coordenadas indicadas.
    - *Punto2D create().* Devuelve el punto con coordenadas (0.,0.).
    - *Punto2D create(Punto2D p).* Crea un punto con las mismas coordenadas que p.
    - *Punto2D create(String s).* Crea un punto a partir de su representación como cadena de caracteres.
- *Casos de prueba*
    - ….

A partir del esquema anterior los métodos del tipo serían

- *Double getX().*
- *void setX(Double x).*
- *Double getY().*
- *void setY(Double y).*
- *Punto2D getOrigen(),*
- *Double getDistanciaA(Punto2D p).*
- *Double getDistanciaAlOrigen()*
- *void mueveA(Double x, Double y)*

Además suponemos disponibles los métodos, heredados de Object, que definen la igualdad y características relacionadas.

- *boolean equals(Object p).*
- *int hashCode().*
- *String toString().*
- *int compareTo(Punto2D p)* (porque lo hemos dotado de orden natural)
- *int compareByX(Punto2D p)*
- *int compareByY(Punto2D p)*
- *int compareByDistanciaAlOrigen(Punto2D p)*

Y los métodos de la factoría

- *Punto2D create(Double x, Double y).*
- *Punto2D create().*
- *Punto2D create(Punto2D p).*

- *Punto2D create(String s).*

Algunos Casos de prueba para el tipo Punto2D son:

Método	this	p	r	this´	Excepción
create		(2.,3.)	(2.,3.)		
create		"(-2.,7.)"	(-2.,7.)		
create		"(===)"			*IllegalArgumentException*
create		()	(0.,0.)		
getX	(-5.,9.)		-5.		
getY	(6.,3.)		3.		
getDistanciaA	(1.,1.)	(2.,2.)	`1.4142`		
equals	(6.,9.)	(6.,9.)	true		
equals	(6.,9.)	(5.,9.)	false		
compareTo	(7.,4.)	(6.,8.)	+1		
compareTo	(2.,3.)	(10.,30.)	-1		
compareTo	(2.,3.)	(2.,3.)	0		
mueveA	(1.,4.)	(3.,4.)		(3.,4.)	

2. Factorías

El enfoque que hemos visto hasta ahora para crear objetos se puede resumir en el ejemplo:

Racional r1 = new RacionalImpl();

Es decir declaramos una variable de un tipo (normalmente un interface) e inicializamos el objeto mediante una llamada a un constructor de una clase concreta precedido del operador new. Frente a la forma anterior es conveniente usar otro diseño que oculte el mecanismo de creación de los objetos. Una factoría es una solución para ese problema. Una **factoría** es un patrón que nos permite **construir objetos** de uno o varios tipos, **ocultando los detalles de las clases concretas que implementan el o los tipos**. Cada factoría es un tipo nuevo con métodos adecuados para crear objetos. El diseño de ese tipo se hará pensando en la funcionalidad que necesiten los posibles clientes de la factoría. Este nuevo tipo lo implementamos directamente como una clase que ofrezca los métodos necesarios para construir los objetos y los implemente.

Un primer ejemplo sencillo es una Factoría de objetos de tipo Racional (fracción):

```
Factoría de Racionales {

      Racional create(String s);  // crea el racional representado por s

      Racional create(Integer a); // crea el racional representado por a/1

      Racional create();          // crea el racional representado por 0/1
```

```
        Racional create(Integer a, Integer b);

                                    // crea el racional representado por a/b
        Racional create(Racional r);

                                // crea un racional igual a r pero no idéntico
        ...
}
```

Las factorías pueden ser de utilidad, también, para gestionar algunas propiedades de la población del tipo. Por ejemplo la factoría puede dotarse de propiedades como: número de racionales creados, menor número racional creado, etc.

Otro uso muy frecuente de factorías es usarlas para disponer de un objeto único que tiene que ser compartido por distintas partes de un programa. Es el equivalente a las llamadas variables globales en lenguajes de programación estructurados. Este tipo de factoría se conoce como un **singleton.** Esto lo podemos concretar en dos métodos: uno que inicialice esa variable a compartir y otro que obtenga el objeto a compartir. Veamos un ejemplo en el cual queremos compartir un valor concreto de tipo Racional.

La factoría de racionales quedaría ampliada con:

```
Factoría de Racionales {

        ...

        Integer getNumeroDeRacionalesCreados();

                // devuelve el número de racionales creados

        Racional getMenorRacional();

                // devuelve el menor de los racionales creados

        void setValorCompartidoInicial(Integer a, Integer b);

                // inicializa el numerador y denominador del racional a compartir

        Racional getRacionalCompartido(); //

                // devuelve el racional compartido. Este será idéntico en todas
                // las llamadas

        ...
}
```

Otro uso adecuado de las factorías es cuando tenemos un tipo con varias implementaciones: un único interface y varias clases implementadoras. O cuando queremos crear objetos de un tipo a partir de objetos de otro. En general es conveniente usar factorías para concentrar en un punto en el programa el mecanismo de creación de los objetos y los posibles cambios en el mismo. Un ejemplo de factoría de listas:

```
Factoría de Listas {

       <T> List<T> newArrayList();

       <T> List<T> newArrayList(T... elem);

       <T> List<T> newArrayList(T[] elem);

       <T> List<T> newLinkedList();

       <T> List<T> create(List<T> elem);

       <T> List<T> reverse(List<T> elem);

       <T> List<T> nCopias(Integer n, T e);

       ...
}
```

Los nombres de los métodos de la factoría anterior indican con cierta claridad la funcionalidad de cada uno de ellos.

Las factorías pueden tener una funcionalidad más compleja. Por ejemplo guardar los objetos creados de un tipo, recuperar de ellos los que cumplan un criterio, recuperar el que tenga una clave dada o crear un nuevo. Veremos en capítulos posteriores algunas de esas posibilidades.

3. Definición de la Igualdad

Una cuestión importante, cuando se diseña un tipo, es la definición de la igualdad entre dos objetos de ese tipo. La igualdad define una clase de equivalencia entre los objetos de un tipo. Son iguales todos los objetos que pertenezcan a una clase dada.

Definida la igualdad hay otras características que están relacionadas: el *hashCode*, la representación como cadena y, si lo hay, el orden natural. Los tipos *Object*, *Equivalence* y *Comparable* nos proporcionan los métodos adecuados para diseñar todos los elementos relacionados con la igualdad. Los métodos púbicos de *Object* ya están disponibles en todas las clases, porque todas heredan de la clase *Object*. Podemos usar el tipo *Equivalence*, del entorno *Guava*, para definir clases de equivalencia. Por otra parte los tipos que tengan orden natural extenderán el tipo *Comparable*. Una variante es el tipo *Comparator* que no permite definir otros órdenes distintos al natural. Veamos cada uno de ellos, sus métodos y los requisitos de los mismos.

3.1 El tipo Object

En Java existe una clase especial llamada *Object*. Como todas las clases tiene asociado un tipo: el tipo *Object*. Todas las clases que definamos y las ya definidas heredan de *Object*. Los

métodos de este tipo son los métodos públicos no *static* del mismo. Veamos en primer lugar algunos de esos métodos públicos: *equals, hashCode y toString*. Aprenderemos sus propiedades, las restricciones entre ellos y la forma de rediseñarlos para que se ajusten a nuestras necesidades. El diseño de estos métodos, para un tipo dado, es la concreción de la definición que demos para la igualdad entre objetos del mismo.

La signatura de estos métodos es:

```
boolean equals(Object o);
int hashCode();
String toString();
```

Como el tipo *Object* es ofrecido por todos los objetos que creemos, los métodos anteriores están disponibles en todos los objetos. Existen otros métodos públicos de *Object* pero no los veremos por ahora.

- El método *equals(Object o)* se utiliza para decidir si el objeto es igual al que se le pasa como parámetro. Recordamos que para decidir si dos objetos son idénticos se usa el operador ==.
- El método *hashCode()* devuelve un entero que es el código *hash* del objeto. Todo objeto tiene, por lo tanto, un código hash asociado.
- El método *toString()* devuelve una cadena de texto que es la representación exterior del objeto. Cuando el objeto se imprima en la pantalla se mostrará como indique su método *toString* correspondiente.

Todos los objetos ofrecen estos tres métodos. Por lo tanto es necesaria una buena comprensión de sus propiedades y un buen diseño de los mismos.

De manera general cuando estamos implementado una clase en Java designamos por *this* el objeto en que estamos. Así el método *equals(r)* devuelve true si *this* (el objeto actual) es igual a *r* que se ha pasado como parámetro.

Propiedades y restricciones:

La primera propiedad importante es que ninguno de los tres métodos puede disparar excepciones. O dicho de otro modo, su precondición es *true* y por lo tanto siempre funcionan en modo normal. Un método funciona en modo normal cuando acaba sin disparar excepciones. Un método con precondición *true* debe funcionar en modo normal para todos los valores de los parámetros de entrada y de su estado. Además hay otras restricciones. Usamos, en los que sigue, el símbolo *=>* para representar el **operador implica** (no disponible directamente en Java).

Propiedades de equals:

- **Reflexiva:** Un objeto es igual a sí mismo. Es decir para cualquier objeto *x* distinto de *null* se debe cumplir *x.equals(x)* es *true* y *x.equals(null)* es *false.*
- **Simétrica**: Si un objeto es igual a otro, el segundo también es igual al primero. Es decir para dos objetos cualquiera *x, y* distintos de *null* se debe cumplir *x.equals(y) => y.equals(x).* Múltiples invocaciones de *x.equals(y)* deben devolver el mismo resultado si el estado de *x* e *y* no han cambiado.
- **Transitiva:** Si un objeto es igual a otro, y este segundo es igual a un tercero, el primero también será igual al tercero. Es decir para tres objetos *x, y, z* distintos de *null* se debe cumplir *x.equals(y)&& y.equals(z) => x.equals(z).*

Propiedades de equals/toString:

- Si dos objetos son iguales, sus representaciones en forma de cadena también deben serlo. Es decir para dos objetos cualquiera *x, y* distintos de *null* se debe cumplir *x.equals(y) => x.toString().equals(y.toString()).*

Propiedades de equals/hashCode:

- Si dos objetos son iguales, sus códigos hash tienen que coincidir. La inversa no tiene por qué ser cierta. Es decir para dos objetos cualquiera *x, y* distintos de *null* se debe cumplir *x.equals(y) => x.hashCode() == y.hashCode().* Sin embargo no se exige que dos objetos no iguales produzcan códigos hash desiguales aunque hay que ser consciente que se puede ganar mucho en eficiencia si en la mayoría de los casos objetos distintos tienen códigos hash distintos.

Las propiedades y restricciones anteriores son muy importantes. Si no se cumplen, el programa que diseñemos puede que no funcione adecuadamente. *To*dos los tipos ofrecidos en *la API de Java tienen un diseño adecuado* de esos tres métodos. Pero recordemos que esos tres métodos están sólo disponibles en los tipos que extienden *Object*. Es decir en los tipos que definen objetos y no están disponibles en los tipos primitivos. Por lo tanto esos métodos ya están disponibles en *Integer, Long, Float, Double y String*, pero no en los tipos *int, long, float y double.*

Las distintas posibilidades de igualdad son casos particulares de relaciones de equivalencia. Para diseñar una relación de equivalencia sobre un tipo escogemos una expresión lógica, definida sobre las propiedades del tipo, que indique cuando dos objetos son equivalentes.

Algunos detalles más hay que considerar para decidir completamente la implementación de la igualdad cuando tenemos un interface *I* implementado por varias clases *C1, C2, C3*. El interface *I* y las clases *C1, C2, C3* son tipos diferentes aunque los tipos de las clases son subtipos del tipo definido por el interface. Un objeto concreto puede ofrecer los tipos I y C1 por ejemplo, otro el I y C2, etc. Aquí la cuestión que aparece es: dos objetos o1, o2 de las clases C1 y C2 que ofrecen los mismos valores para las propiedades relevantes (desde el punto de vista del interface I) son iguales o distintos. Este es un problema de diseño. Podemos considerar iguales dos objetos o1 y o2 que siendo de clases distintas ofrezcan el mismo interface I y las

propiedades relevantes sean iguales. Alternativamente podemos definir una relación de equivalencia más estricta y considerar distintos dos objetos de clases distintas aunque ofrezcan el mismos interfaces y los mismos valores para las propiedades relevantes. Los entornos de ayuda a la programación como Eclipse ofrecen las dos posibilidades aunque las segunda es la más usada.

3.2 El tipo Comparable

Otro concepto relacionado con la igualdad es el orden natural de un tipo. No todos los tipos tienen orden natural pero en general es adecuado, siempre que sea posible, diseñar un orden natural para los objetos de un tipo.

El tipo *Comparable<T>* representa el orden natural sobre objetos de tipo T. Los tipos anteriores definidos como sigue:

```
package java.lang;
public interface Comparable<T>{
    int compareTo(T o);
}
```

El tipo *Comparable<T>* está definido en Java en el paquete *java.lang* y se compone de un sólo método: el método *compareTo*. El tipo Comparable es un tipo genérico que sirve para establecer el orden natural de un tipo dado.

El orden natural (*Comparable*) compara el objeto *this* con otro que toma como parámetro el método *compareTo*. El método *compareTo* devuelve un entero negativo, cero o positivo según que *this* menor, igual o mayor que se pasa como parámetro.

Tiene un conjunto de requisitos:

- El método *compareTo* debe disparar la excepción `NullPointerException` cuando toma como parámetro un valor *null.*
- Los órdenes natural debe ser consistente con la igualdad definida para ese tipo en el sentido de que `e1.compareTo(e2) == 0` debe tener para todos los pares `e1, e2` distintos de *null* el mismo valor que `e1.equals(e2).` Esta propiedad impone restricciones en su diseño.

Junto con el orden natural podemos definir otros órdenes sobre los objetos de un tipo. Lo hacemos con métodos que devuelven un *int*. Los métodos pueden ser diseñados para comparar *this* con otro objeto o para comparar dos objetos *o1, o2* en general. Un ejemplo podría ser el método *int compareByX(Punto2D p).* Este método podría ser adecuado para

representar un orden que ordenara los puntos según el valor de la propiedad X. Estos ordenes distintos del natural los llamaremos órdenes alternativos.

Cada orden, natural o alternativo, define implícitamente una relación de equivalencia. Según esa relación de equivalencia dos objetos son equivalentes si el resultado de su comparación es 0. El hecho de que **una relación de orden sea consistente con la igualdad** implica que las clases de equivalencia definidas por el orden natural y por el método *equals* son las mismas. Por el contrario, si el orden no es consistente con la igualdad, las clases de equivalencia definidas por el orden y por la igualdad pueden ser distintas. Como hemos visto la igualdad de un tipo define una relación de equivalencia que es la misma relación de equivalencia definida por el orden natural. Por cada relación de equivalencia es conveniente escoger, por razones de implementación posteriores, un representante canónico.

Los racionales tienen un orden natural. El racional *a/b* es menor que *c/d* si *a*d* es menor que *b*c*. La relación de equivalencia definida, la misma definida por la igualdad define unas clases. Dos racionales, *a/b* y *c/d,* están en la misma clase si *ad=bc*. Un representante canónico puede ser el racional equivalente simplificado y con denominador positivo.

Propiedades y restricciones:

- *compareTo* y *compareBy…* comparan dos objetos *p1* y *p2* (en el caso de *compareTo p1* es *this*) y devuelve un entero que es:
    - Negativo si *p1* es menor que *p2*
    - Cero si *p1* es igual a *p2*
    - Positivo si *p1* es mayor que *p2*

- *equals/compareTo:* Si el orden definido debe ser coherente con la definición de igualdad tal como se ha explicado antes entonces:
    - El orden natural los diseñamos para que sea consistente con la igualdad. Si *equals* devuelve true *compareTo* debe devolver cero. Aquí también incluimos, tal como se recomienda en la documentación de Java, la inversa. Es decir que si *compareTo* devuelve cero entonces *equals* devuelve true. Por lo tanto con estos requisitos la relación de equivalencia que define la igualdad es la misma que la definida implícitamente por el orden natural. Esto lo podemos enunciar diciendo que la

expresión siguiente es verdadera para cualquier par de objetos *x, y*: `(x.compareTo(y) == 0)==(x.equals(y))`.

- El orden alternativo, definido por métodos del tipo compareBy… no tiene que ser consistente con la igualdad. Si el orden alternativo no es consistente con la igualdad define, además de la relación de orden, unas nuevas relaciones de equivalencia distintas a las definidas por la igualdad. Esto será importante más adelante.

- Propiedades de las relaciones de orden:
  - **Antisimetría**: Si un objeto es menor que otro el segundo es mayor que el primero. Una forma de enunciar esta propiedad para el caso del orden natural es. Esto lo podemos enunciar para cualquier par de objetos x, y con un orden natural como `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`. Y para un orden alternativo cualquiera `cmp` como `sgn(cmp.compare(x,y)) == -sgn(cmp.compare(y,x))`. Donde *sgn* es la función signo.

  - **Transitividad:** Si un objeto es menor o igual que un segundo y este menor o igual que un tercero el primero es menor o igual que el tercero.
    Para tres objetos de un tipo con orden natural debe ser válido

    ```
    x.compareTo(y)<=0) && y.compareTo(z))<=0) ->
                                    x.compareTo(z)<=0)
    ```

    Para tres objetos de un tipo `dado` y un orden *cmp* sobre el mismo debe ser válido:

    ```
    cmp.compare(x,y)<=0) && cmp.compare(y,z)<=0 ->
                                    cmp.compare (x,z)<=0
    ```

- Recomendaciones:

  - La implementación del método *compareTo* debería involucrar sólo las propiedades que participan en definición de la igualdad o propiedades derivadas de las mismas. Además para que el orden definido sea consistente con la igualdad las propiedades consideradas deberían ser evaluados sobre el representante canónico de la clase de equivalencia correspondiente.

4. Contratos asociado a un tipo

Hasta ahora, hemos diseñado los tipos mediante una definición más o menos ambigua de los mismos en lenguaje natural.

Esta forma de proceder presenta algunos problemas:

- Las ambigüedades inherentes al lenguaje natural ocasionan malas interpretaciones de las características de los tipos a implementar.
- Quien solicita la implementación de un tipo a un programador no puede estar seguro de que la clase que le entregue el programador hace exactamente lo que él espera.
- No existe una manera sistemática de realizar pruebas a la clase obtenida que nos aseguren que la implementación es correcta según lo esperado.

En la medida de lo posible es necesario usar conceptos que precisen la definición del tipo. A conjunto de ideas que definen el modo de funcionar de un tipo lo llamamos **contrato del tipo**. Por todo esto, antes de implementarlo, es conveniente definir su **contrato**. Un **contrato** es un documento en el que se define la **funcionalidad** ofrecida por **un tipo** (o un conjunto de tipos) y por lo tanto el uso de la misma por parte de sus posibles clientes.

Un contrato está formado por:

- Métodos del tipo o conjunto de tipos:
    - Definen las operaciones disponibles en el tipo o conjunto de tipos (mediante interfaces o clases)
- **Restricciones**
    - Establecen limitaciones al uso de las operaciones disponibles y ligaduras entre unas operaciones y otras.
    - Se incluyen en el contrato mediante comentarios en lenguaje natural y cuando sea posible mediante expresiones que puedan ser evaluadas.

Existen dos maneras de describir las **restricciones**:

- Mediante **ejemplos**
    - Para cada operación, se describen ejemplos del funcionamiento esperado
- Mediante **expresiones lógicas**
    - Precondiciones, postcondiciones, excepciones, invariantes y estado inicial

Mediante un contrato definimos un tipo o un conjunto de tipos. La parte sintáctica de cada uno de los tipos se convertirá en la interfaz o en la parte pública de la clase que define el tipo. Mediante las restricciones y/o los ejemplos de funcionamiento definimos la semántica de cada uno de los tipos.

4.1 Conceptos previos

Un contrato debería ser completo en el sentido de contener los métodos necesarios para poder especificar las propiedades basándose sólo en los métodos del mismo.

Un contrato puede usar un conjunto de tipos especificados en otro contrato. Diremos que son **tipos usados** por el contrato. A los tipos cuyo contrato que diseñar los llamaremos tipos participantes en el contrato.

Los métodos disponibles en un contrato los clasificamos en **observadores** o **modificadores**. Un método **observador** es aquel cuya ejecución no cambia los valores devueltos por el resto de métodos observadores. Un método **modificador** es aquel cuya ejecución cambia los valores devueltos por alguno de los métodos observadores. Los métodos observadores nos dan acceso a las propiedades de un objeto. Son los mecanismos para observar el estado del mismo. Por lo tanto, por cada propiedad de un objeto tendremos un método observador.

Por defecto consideramos que un método es modificador. Indicaremos que un método es observador con @observador y modificador con @modificador.

Las expresiones pueden ser de dos tipos: **expresiones sin efectos laterales** y **con efectos laterales**. Una expresión se dice que tiene efectos laterales, en el contexto de un contrato, si alguna variable de la expresión cambia de valor al ser evaluada o cambia el valor devuelto por algún método observador. Si alguna variable es de tipo objeto entonces cambiar su valor implica cambiar alguna de sus propiedades. Si ninguna variable cambia de valor al evaluarse la expresión (ni tampoco cambian los valores devueltos por los métodos observadores) decimos que es una expresión sin efectos laterales. Las expresiones ***sin efectos laterales*** del contrato estarán, por lo tanto, constituidas por:

- Constantes y variables de tipos usados o de los tipos participantes en el contrato.
- **Operadores** de Java

- Llamadas a **métodos observadores** del propio contrato o de otros tipos no participantes en el contrato pero usados en él.

Tal como vimos en el tema 2, un método tiene dos modos de funcionamiento que dependen de los parámetros reales que reciba y las propiedades del objeto sobre el que se invoca el método. En el **modo normal** el método termina sin disparar excepciones y devolviendo el resultado adecuado. En el **modo excepcional** el método dispara una excepción y termina. Hay una tercera posibilidad y es que el método no termine. Si eso ocurre hay un problema de diseño en el código del método. El contrato debe especificar la región de funcionamiento en modo normal y en modo excepcional. En el contexto de un contrato admitimos la posibilidad de una **región no especificada.** En esta última región no se indican restricciones sobre el funcionamiento de los métodos. Un método cuya región no especificada sea nula diremos que está completamente especificado.

El contrato debe indicar para cada método la zona de funcionamiento normal y la excepcional.

4.2 Definición de restricciones mediante expresiones lógicas

En esta forma expresamos las restricciones mediante expresiones lógicas ampliadas sin efectos laterales. En muchos casos estas expresiones lógicas y los conceptos relacionados los expresamos en lenguaje natural. En otros casos en algún tipo de lenguaje como Java o alguna extensión del mismo.

Decimos expresiones ampliadas porque permitimos un operador que no aparece en Java. Es el operador @pre. Es un operador sufijo que se aplica sobre una expresión y devuelve el valor de esta expresión calculado antes de invocar al método dado. Por ejemplo:

```
getSaldo()@pre
```

Es el valor devuelto por *getSaldo()* **antes** de la ejecución del método. También tenemos disponible la variable *@return* que contiene el valor devuelto por el método.

Como hemos explicado arriba un contrato se compone de un conjunto de métodos que definen los tipos participantes en el contrato y un conjunto de restricciones. Se incluyen en el contrato una serie de expresiones lógicas que deben cumplirse bajo determinadas premisas.

Los tipos de restricciones son:

- Invariantes
- Precondiciones
- Postcondiciones
- Condiciones de Disparo Excepciones

Los invariantes son restricciones que afectan a todo el contrato, mientras que las precondiciones, postcondiciones y excepciones son específicas para cada método.

Invariante de un contrato es una expresión lógica que debe cumplirse en todo momento. Por lo tanto, antes y después de la invocación de cada método y después de la ejecución de cada uno de los constructores. Lo definimos con @invariante seguido de una expresión. Tiene como ámbito todo el contrato.

Precondición de un método**.** Es una expresión lógica que especifica la región en que el método trabaja en modo normal. Es una expresión lógica que involucra los parámetros reales de la llamada al método y los valores de las propiedades del objeto sobre el que se invoca. Si cuando llamamos a un método no se cumple la precondición el funcionamiento del método no estará especificado o disparará una excepción si así se indica en la sección correspondiente. La definimos con @pre seguido de una expresión. Tiene como ámbito el método en el que se define. Si un método no tiene especificada una precondición se asume que ésta es siempre verdadera.

Postcondición de un método. Es una expresión lógica ampliada que debe ser válida cuando el método termina, supuesto que ha sido invocado cumpliendo su precondición. La postcondición involucra los parámetros reales, el valor de retorno y las propiedades del objeto antes y después de la invocación del método. La definimos con *@pos* seguido de una expresión. Tiene como ámbito el método en el que se define. Si un método no tiene especificada una postcondición se asume que ésta es siempre verdadera

Condiciones de Disparo de Excepciones en un método**.** Son un conjunto de expresiones lógicas, construidas sobre las propiedades del objeto y los parámetros reales que indican las condiciones en las que se lanzarán excepciones, y qué excepción lanzará. La combinación mediante el operador *OR* de las condiciones de disparo de excepciones define la región de comportamiento excepcional. La definimos con *@excepcion* seguido de una expresión y la

excepción disparada separadas por *throw*. Tiene como ámbito el método en el que se define. Junto con las excepciones especificadas se puede disparar la excepción *NoSeCumpleElContrato* cuando se violen algunas de las restricciones especificadas.

Ejemplo de contrato, con restricciones definidas mediante expresiones lógicas:

```
//     @Contrato Cuenta extiende Object,Comparable
//     @invariante getSaldo() >= 0

tipo Cuenta extends Comparable<Cuenta> {
       //  @observador
       Double getSaldo();

       //  @pre: c > 0 && c <= getSaldo()
       //  @pos: getSaldo().equals(getSaldo()@pre – c)
       //  @excepcion:  c <= 0  throw IllegalArgumentException ()
       //  @excepcion:  c > getSaldo() throw SaldoInsuficienteException()
       //  @modificador
       void retirar(Double c);

       //  @pre: c > 0
       //  @pos: getSaldo().equals(getSaldo()@pre + c)
       //  @excepción:  c<=0  throw IllegalArgumentException ()
       //  @modificador
       void ingresar(Double c);
}
```

Invariante: En todo momento de la vida del objeto, el saldo será mayor o igual a cero.

```
//     @invariante getSaldo() >= 0
```

@observador

Indica que se trata de un método observador, es decir, que su invocación no modifica los valores de las propiedades del objeto.

```
tipo Cuenta {
       //  @observador@
       Double getSaldo();
```

Precondición: El parámetro *c* debe ser mayor de cero, y la propiedad *saldo* mayor o igual que *c*.

Postcondición: En caso de invocar al método respetando la precondición, el saldo de la cuenta se verá rebajado en un valor igual al parámetro *c*.

```
//  pre: c > 0 && getSaldo()>=c
//  pos: getSaldo().equals(getSaldo()@pre – c)
```

Excepción: En caso de recibir un valor negativo o cero del parámetro *c*, el método lanzará una excepción del tipo ***IllegalArgumentException***.

@modificador

Indica que es un método modificador, es decir, que su invocación puede modificar el estado interno del objeto.

```
//  @excepción:  c <=0  throw IllegalArgumentException ()
//  @excepcion:  c > getSaldo() throw SaldoInsuficiente()
//  @modificador@
```

4.3 Zona de funcionamiento normal, de funcionamiento excepcional y no definida

Si *pre* es la precondición y *cd1, cd2, …, cdn,* las condiciones de disparo de las diferentes excepciones entonces la zona de funcionamiento normal viene definida por *pre*, la zona de funcionamiento excepcional por *cde1 ||cde2|| … cdn.* La expresión lógica para la zona no definida es *!(pre || cde1 ||cde2||…).* Además en todo momento es válido el invariante. Es decir la expresiones que definen las tres zonas están reforzadas (combinadas con el operador &&) con el invariante.

En el método *retirar* la zona de funcionamiento normal está definida por la expresión lógica: `c > 0 && c <= getSaldo()`. La zona de funcionamiento excepcional está definida por `c <= 0 || c > getSaldo()`. Podemos comprobar que el método está completamente especificado porque la zona no definida está vacía.

En el método *ingresar* la zona de funcionamiento normal está definida por la expresión lógica: `c > 0`. La zona de funcionamiento excepcional está definida por `c <= 0`. Podemos

comprobar que el método está completamente especificado porque la zona no definida está vacía.

4.4 Métodos observadores, precondiciones, postcondiciones e invariantes

Los métodos observadores pueden tener precondiciones. Esto quiere decir que la correspondiente propiedad sólo está definida cuando la precondición es verdadera. Si se invoca un método observador en una situación donde no se cumple la precondición entonces se disparará una excepción como en el resto de los métodos. A partir de los métodos observadores con precondiciones se pueden construir expresiones pero estas no se pueden evaluar (disparan una excepción) si no se cumple la precondición.

Los métodos observadores pueden estar ligados por restricciones que son válidas en todo momento. Estas restricciones se expresan mediante invariantes. Pero una forma más adecuada, si es posible, es la siguiente:

- Clasificar los métodos observadores en básicos y derivados (según estén asociados a una propiedad básica o derivada).
- Los métodos observadores básicos no tienen postcondición.
- Los métodos observadores derivados sí tienen postcondición. Ésta se escribe como una expresión que usa los métodos observadores básicos, u otros métodos observadores cuya postcondición hayamos escrito previamente.
- Si existen restricciones adicionales entre los métodos observadores que no puedan ser incluidas en las postcondiciones anteriores se ponen en el invariante.

4.5 Contratos inconsistentes o no implementables

En un contrato pueden aparecer inconsistencias de varios tipos que tiene que ser evitadas:

- Una precondición es siempre falsa. En ese caso el método que estamos especificando no podrá ser invocado nunca.
- Dos condiciones de disparo de dos excepciones diferentes son válidas a la vez.
- La condición de disparo de una excepción es válida en algunos casos junto con la precondición.

Por otra parte, la especificación de un método puede ser no implementable para un subconjunto de los valores de los parámetros reales y propiedades del objeto. Debemos tener en cuenta que antes de la invocación de un método deber ser válida la expresión *Invariante*

&& Precondicion y después de la invocación *Invariante && Postcondición.* Después de invocados los constructores debe ser válida la expresión *Invariante.*

Sea el contrato (modificado del anterior) siguiente.

```
//     @invariante getSaldo() >= 0
//     @inicial    getSaldo() > 0

public interface Cuenta {
       //  @observador
       Double getSaldo();

       //  @pre: c > 0
       //  @pos: getSaldo() == getSaldo()@pre - c
       //  @excepcion:  c<=0  throw IllegalArgumentException
       //  @excepcion:  c>getSaldo() throw SaldoInsuficiente()
       //  @modificador
       void retirar(Double c);
```

Antes de invocar el método retirar es verdadero *getSaldo()>=0 && c>0*. Después de invocar el método debe ser válido *(getSaldo() == getSaldo()@pre – c) && (getSaldo()>=0).* Si invocamos el método con un parámetro *c > getSaldo()* la precondición es válida. El método puede ser invocado. Pero es imposible implementar el método para conseguir que al finalizar sea válida la expresión exigida. En este caso el implementador debe disparar una excepción que no estaba especificada debido a que el contrato es no implementable.

La solución, en el caso anterior, es reforzar la precondición para que sean permitidos sólo valores de los parámetros reales y la precondiciones que hagan posible implementar el método.

Pero en muchos casos es difícil detectar conjuntos de valores de los parámetros y las propiedades que hagan inviable el contrato. Si el contrato es inviable el implementador debe disparar una excepción que no estaba especificada.

4.6 Definición de contratos mediante ejemplos de funcionamiento

En la mayoría de las situaciones se añaden restricciones adicionales al contrato del tipo mediante un conjunto de ejemplos de funcionamiento. Cada ejemplo incluye valores para los parámetros y las propiedades del objeto antes de la llamada, y el **valor esperado** de las propiedades del objeto después de la ejecución del método y también el valor de devolución esperado si lo hay.

Para cada método de un tipo podemos añadir conjunto de ejemplos de funcionamiento que capture lo mejor posible la casuística de utilización del método y que sirva para completar, si las hay, las restricciones del contrato.

Especificar el contrato de un tipo mediante ejemplos de funcionamiento suele ser una práctica habitual y fácil de usar. Pero en algunos casos es más claro y completo especificar las restricciones del contrato mediante expresiones lógicas tal como hemos visto anteriormente. Mediante expresiones lógicas especificamos el funcionamiento de un método para conjuntos de valores. La combinación de ambas técnicas: ejemplos de funcionamiento y restricciones expresadas mediante expresiones lógicas puede ser adecuado. Usaremos cada técnica cuando pueda ser útil.

4.7 Otras restricciones más generales en el contrato de un tipo

Hay algunas propiedades interesantes que no se pueden expresarse con las restricciones que hemos permitido hasta ahora. Son, por ejemplo, las restricciones del contrato asociado a la igualdad y las de *Comparable*. Una de ellas es la propiedad transitiva de la igualdad:

$$\forall\, T\ o1, o2, o3 : Object : o1.equals(o2)\ \&\&\ o2.equals(o3) \Rightarrow o1.equals(o3)$$

Para incluirla en el contrato necesitamos ampliar el tipo de restricciones disponibles. Anteriormente las restricciones que hemos usado sólo han sido expresiones lógicas sin efectos laterales. Ahora vamos a permitir un segundo tipo de restricciones: secuencia de sentencias (posiblemente con efectos laterales) terminadas con una expresión lógica y en las que permitimos el cuantificador para todo y algunos operadores más como el operador de implicación. Este tipo de restricciones las incluimos como parte del invariante. La forma concreta de escribir la expresión anterior en un contrato es:

```
// @inv Object o1, o2, o3; {o1.equals(o2) && o2.equals(o3) => o1.equals(o3);}
```

Donde asumimos que las variables declaradas están cuantificadas universalmente y hemos usado por claridad el operador implica (=>). Un segundo ejemplo sería expresar que si de una cuenta sacamos una cantidad, y luego ingresamos la misma cantidad, la cuenta queda igual:

```
// @inv Cuenta c1, c2; Double x;
          {c2 = c1.clone(); c1.retirar(x); c1.ingresar(x); c1.equals(c2);}
```

La última sentencia de estas restricciones más generales (formadas por una secuencia de sentencias) deberá ser una expresión lógica. El valor lógico de esta expresión (*true* o *false*) nos indicará si la restricción es verdadera o falsa. Las variables declaradas al principio (antes de {…}) se suponen cuantificadas universalmente.

La restricción anterior podría incluirse en el contrato *Cuenta*. Hemos asumido que ya está previamente definido el contrato de la igualdad (que define *equals*, *hashCode*, *toString*) y basado en él el contrato *Comparable* (que define *compareTo*). Las propiedades de estos contratos se han visto arriba.

Tanto estas secuencias de sentencias como las expresiones lógicas vistas arriba pueden tener una precondición. Esta precondición es la combinación (con el operador &&) de las precondiciones de los métodos que participan en la secuencia de sentencias o expresión lógica. Esto es importante porque si no se cumple la precondición no hay valor (verdadero o falso) para la restricción porque se dispara una excepción al intentar evaluarla.

Con estas restricciones más generales el contrato de la igualdad puede definirse como:

```
//@Contrato Object
//@inv Object o          {o.equals(o);}
//@inv Object o1,o2      {o1.equals(o2) => o2.equals(o1);}
//@inv Object o1,o2,o3   {o1.equals(o2) && o2.equals(o3) => o1.equals(o3);}
//@inv Object o1, o2     {o1.equals(o2) => o1.hashCode() == o2.hashCode();}
//@inv Object o1, o2     {o1.equals(o2) => o1.toString().equals(o2.toString();}
```

Este tipo de restricciones nos permiten obtener casos de prueba que nos permitirán comprobar su correcto funcionamiento.

5. Tipos y Subtipos

Entre los tipos existen distintas relaciones:

- **Uso**: Un tipo T1 usa otro T2 cuando en el contrato de T1 aparece alguna expresión o variable cuyo tipo es T2.
- **Subtipado**: La relación de subtipado es más compleja cuando consideramos el contrato del tipo.

Desde el punto de vista sintáctico si S es un subtipo de T entonces ofrece todos los métodos de T y posiblemente algunos más pero no menos. Como hemos visto en capítulos anteriores en Java hay dos formas de definir tipos: tipos *interfaces, y clases (*los tipos *enum* podemos considerarlos un tipo especial de clases). Un tipo S (definido por una interfaz o clase) es un subtipo directo de otro T si:

- Si ambos son definidos por interfaces y *S extends T*.
- Si ambos son definidos por clases y *S extends T*
- Si S es definido por una clase, T por una interfaz y *S implements T*.

En general S es un subtipo de T si es un subtipo directo de él o a través de otros tipos intermedios.

Desde el punto de vista semántico (es decir teniendo en cuenta el contrato del tipo) las condiciones de subtipado son más fuertes. Un tipo S es un subtipo de T cuando un objeto de tipo S puede ser colocado en cualquier punto donde se espera un objeto de tipo T y su funcionamiento será correcto tanto desde el punto de vista sintáctico como semántico. O dicho de otra forma los objetos de tipo T pueden ser remplazados por objetos de tipo S. Esto implica que cualquier propiedad que tenga un objeto de tipo T es también una propiedad de los objetos de tipo S.

Es decir si S es un subtipo de T:

- S ofrece todos los métodos de T y posiblemente algunos más.
- El invariante de S es más fuerte que el de T. Es decir cumple todas las propiedades de T y algunas más
- Para cada método la postcondición de S es igual o más fuerte que la de T.

- Para cada método de S la zona de funcionamiento normal, cuando la precondición es verdadera, debe ser igual o más amplia que la de T.
- S no debe disparar más excepciones que T aunque si puede refinarlas. Es decir si en T se especifica el disparo de una excepción *e* cuando se cumple *cde* entonces en S puede disparar una excepción *e1* que sea un subtipo de *e* en unas condiciones *cde1* más fuertes que *cde.*
- La zona no definida en S es igual o más estrecha que en T.

A partir de un tipo T podemos definir un subtipo S mediante una relación de refinamiento. Esta relación implica definir S a partir del contrato de T (o posiblemente de varios tipos):

- Añadiendo nuevos métodos
- Añadiendo más invariantes.
- A cada método añadiendo más precondiciones y postcondiciones
- Definiendo condiciones reforzadas de disparo de excepciones hijas

Cuando definimos un subtipo S refinando otro T:

- S ofrece todos los métodos de T y los añadidos en la relación de refinamiento.
- El invariante de S es el heredado de T combinado con el operador && con el añadido en la relación de refinamiento
- La precondición de cada método es la heredada de T (si la hay) combinada con el operador || con la añadida en la relación de refinamiento
- La post-condición de cada método es la heredada de T (si la hay) combinada con el operador && con la añadida en la relación de refinamiento
- Las condiciones de disparo son las de T reforzadas en la relación de refinamiento

Las propiedades anteriores son válidas para la igualdad. Tal como hemos visto más arriba la igualdad es una relación de equivalencia sobre los objetos de un tipo. La igualdad definida sobre un subtipo debe ser una relación de equivalencia más fuerte en el sentido de que las clases de equivalencia para el subtipo deben estar incluidas en las clases de equivalencia del tipo padre.

Entre dos relaciones de equivalencia pueden existir relaciones de inclusión. Decimos que una relación de equivalencia es más **estricta** que otra si cada clase de equivalencia de la primera

está incluida en una de la segunda. Decimos que la segunda es más **laxa** que la primera. Desde este punto de vista la relación de equivalencia más estricta posible sobre un tipo dado es aquella en la que dos objetos son equivalentes solamente si son idénticos.

Vemos, pues, que cuando definimos un subtipo *S1* de otro *T* podemos tener dos relaciones de equivalencia: *eq0*, una relación de equivalencia sobre objetos de tipo *T,* y otra *eq1* que define una relación de equivalencia entre objetos de tipo *S1*. Pero todo objeto de tipo *S1* también ofrece el tipo T por lo que tenemos ambas relaciones de equivalencia para definir el método *boolean equals(Object p)* de los objetos de tipo *S1*. Ahora se plantea una cuestión: ¿cuál es la relación de equivalencia adecuada cuando usamos conjuntamente objeto de tipo *S1* y *T*? Los objetos de tipo *S1* pueden ofrecer la relación de equivalencia *eq1* pero también la *eq0*. Supongamos que tenemos un segundo subtipo *S2* que ofrezca la relación de equivalencia *eq2* (más estricta que la *eq0*). Sean los objetos *e1, f1* de tipo *S1,* y *e2, f2* de tipo *S2* y por otra parte *e0*, f0 de tipo *T*. ¿Cuál es la igualdad adecuada para comparar *e1* con *e2*, *e1* con *e0*, *e1* con f1, etc.? En definitiva ¿cuál es la nueva relación de equivalencia que nos permite comparar los objetos de los diversos subtipos y los del tipo? Una posible solución es la siguiente:

- Dos objetos de un tipo *S1* se comparan según la igualdad *eq1* (igualmente con *S1* y *eq2*). Por lo tanto cualquier objeto de tipo S1 es distinto a otro de tipo S2.
- Sean ahora un objeto *e1* de tipo *S1* y otro *e0* de tipo *T*. No olvidemos que los objetos de tipo *S1* son también de tipo *T* por lo tanto podría usarse la relación de equivalencia *eq0* para compararlos. Esto rompería la relación de equivalencia como vamos a ver en un ejemplo más abajo. La solución más simple es que al comparar objeto de tipo *S1* y de tipo *T* se use la relación *eq1* si ambos son de tipo *S1* y en otro caso se consideren distintos.
- En definitiva dos objetos de tipos *S1* y *T* (S1 subtipo de T) sólo pueden ser iguales si ambos son de tipo *S1*. Una versión más estricta aún sería escoger que dos objetos solamente pueden ser iguales si son de la misma clase.

Sean los objetos e1, e0, e2. Si usamos eq0 para comparar e1 con e0 y e0 con e2 puede ocurrir que e1 sea igual a e0 y e0 igual a e2 pero no sean e1 igual a e2. Este problema desaparece con la solución propuesta. Ejemplos y detalles de implementación se darán en el próximo capítulo.

El diseño de tipos con la previsión de que posteriormente puedan ser subtipados requiere de un análisis riguroso que prevea la posibilidad de añadir invariantes y la política de gestión de

esos invariantes. Igualmente requiere un diseño adecuado de la igualdad que tenga en cuenta las consideraciones anteriores.

6. Diseño de tipos y métodos: estilos de programación

Existen dos estilos de programación en este ámbito: uno que hace **Gestión Previa** de las posibles excepciones y otro que hace **Gestión Posterior** de las mismas. Estos estilos, como veremos ahora, no son puros y pueden mezclarse para hacer una programación lo más robusta y comprensible posible.

Como hemos visto en el contrato de un tipo cada posible excepción *ex* que un método puede disparar le podemos asociar una *Condición de Disparo cdex.* Esta condición de disparo debe poder escribirse como una combinación de los parámetros reales del método, las propiedades del objeto (métodos *get* de la interfaz) y operadores. Si eso no fuera posible, porque la expresión dependiera de detalles internos del estado del objeto (atributos), nos está indicando que es necesario un método más en el tipo que, ocultando los detalles del estado del objeto, nos permita escribir la expresión.

Como ya sabemos un método tiene dos modos de comportamiento: normal y excepcional (además puede haber una región indefinida en el contrato). Comportarse de un modo u otro depende de los valores de los parámetros reales y de las propiedades del objeto. Por lo tanto junto con las *Condiciones de Disparo* de las excepciones es conveniente explicitar otra expresión lógica, dependiente igualmente de las propiedades del objeto y de los parámetros reales del método, que nos indique si el método funciona en modo normal. Esta expresión lógica es la que hemos llamado **Precondición**. O dicho de otra forma, si la precondición de un método es verdadera éste funcionará en modo normal y por lo tanto no disparará excepciones. Diseñar adecuadamente las excepciones disparadas por un método, sus Condiciones de Disparo y su Precondición es, en la mayoría de los casos, un proceso que puede ser lento pero necesario.

Si suponemos que el método *m(p)* puede disparar las excepciones *e1, e2* con condiciones de disparo *c1, c2* y precondición *pre,* los estilos de programación citados tienen los siguientes esquemas a la hora de llamar al método *m* sobre el objeto *o* con parámetros reales *r*. El modo de Gestión Previa es de la forma:

```
if(pre(r)){
   o.m(r);
} else if (c1(r){
   código_reparacion_1;
} else if(c2(r)){
   código_reparacion_2;
}
```

Como vemos es un conjunto de *if-else* donde se van evaluando la precondición y las diferentes condiciones de disparo posibles. Si la precondición es verdadera se llama al método. Si, alternativamente (la precondición es falsa), alguna de las condiciones de disparo es verdadera no se llama al método y en su lugar se ejecuta algún código de reparación. Este esquema se simplifica cuando la condición de disparo es *!pre*.

El modo de Gestión Posterior tiene el esquema:

```
try{
    o.m(r);
} catch(TipoE1 e1) {
    código_reparacion_1;
} catch(TipoE2 e2){
    código_reparacion_2;
}
```

Ahora se trata de llamar al método dentro del bloque *try* y si se dispara una excepción gestionarla con la sentencia *try-catch.*

Los dos estilos se diferencian en las estructuras de control usadas para gestionar las posibles excepciones. Hay una diferencia fundamental, además del uso de una estructura de control u otra. En el modo de Gestión Posterior las Condiciones de Disparo no están explícitas, tampoco está explícita la precondición, porque no son necesarias, tal como vemos en el esquema anterior. En muchos diseños, pensados para este modo de gestión de las excepciones, ni siquiera son construibles usando sólo llamadas a métodos consultores, parámetros reales y operadores.

En el estilo de Gestión Previa la Precondición y las Condiciones de Disparo están explícitas, tal como se ve en el esquema, y son usadas para decidir si el método estará en el modo normal o disparará una excepción concreta.

¿Qué método de gestión de la posibles excepciones elegir? ¿Decidimos diseñar las excepciones *e1, e2* como hijas de *RuntimeException* o de *Exception*? Cada método tiene sus ventajas e inconvenientes. Pero hay algunas cuestiones que no se deben olvidar:

- Cuando diseñamos un método para un tipo y pensamos que puede disparar excepciones, entonces por cada excepción disparada debe haber la posibilidad de escribir una condición de disparo en base a los parámetros del método y llamadas a métodos consultores del tipo. Igualmente debe haber la posibilidad de escribir la precondición. Es decir, debemos hacer explícitas las Condiciones de Disparo y la Precondición.
- Si diseñamos las excepciones como hijas de *Exception* entonces tienen que ser declaradas en la signatura del método mediante una cláusula *throws*. Esto nos obliga a usar un estilo de gestión posterior de las excepciones. Por el contrario si diseñamos las excepciones como hijas de *RuntimeException* no tenemos que declararlas en la signatura del método y podemos usar cualquiera de los dos estilos.
- En conclusión si las excepciones son hijas de *RuntimeException* y la precondición y las Condiciones de Disparo son explícitas podemos usar cualquiera de los dos estilos. Si, por el contrario, las excepciones son hijas de *Exception* o las Condiciones de Disparo no son explícitas entonces tenemos que usar la Gestión Posterior de excepciones. Por supuesto podemos mezclar los dos estilos en la forma que consideremos conveniente pero si es posible siempre es recomendable hacer explícitas las Condiciones de Disparo.

Un tema relacionado con el anterior es la metodología para la **Gestión de los Invariantes** y otras propiedades del tipo. Hay dos tipos de políticas para gestionar los invariantes. La elección de una forma de gestionar los invariantes impacta de una forma muy importante en la signatura de los métodos modificadores del tipo.

Las dos políticas son:

- Disparar una excepción para valores de parámetros que violen el invariante.
- Para valores de parámetros que violen el invariante no modificar el estado del objeto y hacer que el método devuelva un valor que informe de ello.

Ambas políticas se pueden combinar de la forma más adecuada a cada problema en particular. Si pensamos un tipo para que a partir de él se puedan diseñar subtipos es muy importante

pensar en sus posibles invariantes y decidir en cada caso la política de gestión de cada uno de ellos.

Veamos el caso concreto del tipo *Collection:* sus posibles invariantes, la política de gestión de los mismos escogida y el impacto sobre el diseño del método *add.*

El tipo *Collection* se diseña para representar agregados de datos de distintos tipos como pueden ser listas, conjuntos, etc. Además se prevé que puede haber colecciones creadas con distintas propiedades: no modificables, que no contengan valores *null*, que todos los elementos de la colección cumplan una propiedad y que en determinados estados de la colección se respeten algunas propiedades adicionales.

Se escoge la política de disparar una excepción si no se cumple alguna propiedad específica especificada en el momento de construir la colección. Así:

- Las colecciones pueden ser modificables y no modificables. Si se construye una colección no modificable y se intenta usar un método modificador se dispara la excepción *UnsupportedOperationException.*
- Las colecciones pueden contener elementos nulos o no. Si se construye una colección que no puede contener elementos nulos y se intenta añadir uno se dispara la excepción *NullPointerException.*
- Una colección puede exigir una propiedad adicional a todos sus elementos. Por ejemplo si es de enteros que todos sean pares. Si construimos una colección cuyos elementos deben cumplir una propiedad e intentamos añadir un elemento que no la cumple se dispara la excepción *IllegalArgumentException.*
- Una colección puede exigir que todos los elementos sean de una clase específica. Si construimos una colección cuyos elementos deben ser de una clase específica e intentamos añadir un elemento que no lo es se dispara la excepción *ClassCastException.*

A su vez se prevé la posibilidad de añadir otros invariantes para construir subtipos. Así el tipo *Set<E>* es un subtipo de *Collection<E>* con los mismos métodos pero un invariante adicional: un conjunto no puede tener dos elementos iguales. Veamos con estas restricciones el diseño escogido para el método *add*. La signatura escogida para el método es:

```
boolean add(E e)
```

La postcondición escogida para este método en el tipo *Collection<E>* obliga a que el elemento añadido pertenezca a la colección después de ejecutar el método. En el subtipo *Set<E>* se añade el invariante mencionado que se decide gestionar sin disparar excepciones. Por eso se prevé que el método devuelva un *boolean* que indica si el elemento ha sido añadido a la colección o no.

7. Casos de prueba a partir de contratos

Una vez definido el contrato asociado a un tipo debemos concretar un conjunto de casos de prueba que nos servirán para comprobar que las implementaciones del tipo cumple el contrato. En muchos casos el contrato se concreta directamente mediante los casos de prueba. Esta segunda posibilidad suele ser la que de forma práctica se usa de forma mayoritaria.

El conjunto de casos de prueba se presenta como una tabla. Cada caso de prueba da lugar a un test. Es decir, una prueba del funcionamiento adecuado del método en un caso concreto

El conjunto de casos de prueba para un método debe capturar lo mejor posible la casuística de utilización del método. Los casos de prueba incluirán los ejemplos de funcionamiento incluidos en el contrato. Además incluirán otros casos de prueba deducidos de las restricciones del contrato.

Por cada método, deducidos de las restricciones del contrato, indicamos casos de prueba que hagan el al método funcionar en modo norma y excepcional. Si los valores del caso de prueba hacen que el método funcione en modo excepcional hay que añadir, en el caso de prueba, la excepción que se espera. Si los valores del caso de prueba hacen que el método funcione en modo normal (no se espera que se disparen excepciones) entonces hay que añadir al caso de prueba los valores esperados devueltos por el método y los valores esperados de los métodos observadores relevantes. Estos valores esperados se pueden deducir, si existen, de las restricciones del contrato.

Las pruebas deben ser diseñadas junto a los contratos. En muchos casos las pruebas son directamente los contratos. Generalmente el diseño de un contrato no es una tarea sencilla y

hay muchos pequeños detalles que se olvidan fácilmente después de poco tiempo. Por ese motivo, lo mejor es desarrollar los contratos y casi en paralelo las pruebas. El diseño del contrato ayudará al de las pruebas y viceversa. Las pruebas deben ser diseñadas por personas no implicadas en la implementación del contrato.

Veamos algunas sugerencias para general un conjunto de pruebas para un tipo cuyo contrato hemos especificado. Como hemos visto cuando diseñamos un tipo es conveniente en muchos casos diseñar una factoría para el mismo. Es conveniente diseñar las pruebas para un tipo junto con su factoría.

La primera idea es que debemos desarrollar un conjunto de pruebas para comprobar el correcto funcionamiento de los métodos del tipo y de los de su factoría.

Por cada método (del tipo y de su factoría) debemos diseñar pruebas con valores que hagan funcionar el método en modo normal y en modo excepcional. En el modo normal para cada conjunto de valores de caso de prueba habrá un posible valor devuelto por el método y unos posibles valores esperados para las propiedades. En el funcionamiento excepcional habrá una excepción esperada. Para que el método funcione en el modo normal tendrá que verificarse una precondición y en el modo excepcional unas condiciones de disparo de una excepción concreta. Ambas, precondición y condiciones de disparo de excepciones son expresiones lógicas cuya estructura nos puede guiar en el diseño de los casos de prueba. Veamos algunas ideas.

Normalmente la precondición (y las condiciones de disparo de excepciones) son la composición de expresiones con operadores lógicos. Generaremos casos de prueba que cubran todas las posibilidades para las expresiones básicas. Por ejemplo, si se tiene una precondición del estilo x > 0 && y > 0 entonces generaremos casos de prueba que hagan:

`x > 0`	`y > 0`
T	T
T	F
F	T
F	F

Esto significa que tenemos que desarrollar al menos un caso de prueba positivo en el que los dos predicados se cumplen y al menos tres casos de prueba negativos en los que alguno de los predicados o varios a la vez son falsos.

En el caso de predicados de la forma @ x in C • P(x), en donde @ puede ser *existe*, *para todo, suma*, etc. y *C* un agregado de elementos hay que diseñar al menos tres casos de prueba: uno en el que la colección *C* no tenga ningún elemento, otro en el que tenga un elemento y otro en el que tenga varios elementos. Por cada uno de los casos anteriores tener en cuenta la estructura del predicado P(x).

Diseñar pruebas específicas para los valores límites. Para ello se analizan todas las expresiones relacionales, se identifican los valores límites y diseñan pruebas específicas para cada valor límite. Los casos de prueba toman el valor límite, el valor del límite más uno y el valor límite menos uno. Por ejemplo, en el caso del predicado x > 0 tendríamos que diseñar una prueba para x == 0, otra para x == 1 y otra para x == -1. En los casos en que una parte del predicado es una constante y la otra una variable el límite está claro, es la constante; en los casos en que las dos partes del predicado son variables es necesario fijar una como si fuera la constante y la otra como variable. Por ejemplo, si tenemos la expresión x > y con x e y de tipo entero, podríamos hacer una prueba con x == y, otra con x == y +1 y otra con x == y - 1. O también una prueba con y == x, otra con y == x +1 y otra con y == x – 1.

En el caso de variables que sean de algún tipo definido por el usuario es conveniente hacer una prueba específica para el caso de que la variable tome el valor *null.*

En el caso de variables de tipo colección (lo que incluye las variables de tipo *String*) debemos hacer al menos una prueba en la que la colección tiene cero elementos, un elemento y varios elementos.

8. Ejecución de los Casos de Prueba

Como hemos comentado anteriormente, al diseñar un tipo nuevo debemos escribir una interfaz y un conjunto de casos de prueba que indicarán la forma en que debe comportarse la

implementación. O dicho de otra forma, los valores esperados (o excepciones esperadas) cuando invocamos un método con determinados parámetros reales y cuando llamamos a un constructor. Estos casos de prueba los reunimos en una tabla como la que hicimos para el tipo Racional.

Los casos de prueba se piensan siempre para una clase dada que implementa un tipo. Un ejemplo es la clase *RacionalImpl* y el tipo *Racional*. Para decidir si la implementación pasa los casos de prueba diseñados podemos usar la herramienta *jUnit*. Otra alternativa es Implementar una clase *TestTipo* (en este caso *TestRacional*) que haga un trabajo equivalente a la herramienta *jUnit* o muestre los resultados esperados en cada uno de los casos de prueba.

Veamos el funcionamiento de la herramienta *jUnit.* Como ejemplo a seguir tomaremos los casos de test especificados para el tipo *Racional*. Los casos de test están especificados para cada método o constructor. Puede haber varios casos de test para un mismo método.

Se trata de implementar una clase denominada *testRacional.* En esta clase se implementará un método de test por cada caso de prueba aunque podemos agrupar varios casos de prueba en el mismos méto El nombre del método comienza por test y se compone del nombre del método Si hay varios casos de prueba para un método dado se le añaden sufijos 1, 2, etc. Cada método está etiquetado con *@Test*.

La herramienta *jUnit* puede generar automáticamente las signaturas de los métodos de test.

El cuerpo de los métodos de test sigue siempre el mismo patrón: se crea un el objeto es un estado dado*,* se le aplica el método indicado y se compara el resultado con el valor esperado con el método *assertEquals (Object, Object).* También es posible usar el método *assertTrue (Boolean).* Este método recibe como parámetro una expresión lógica que compara los resultados producidos con los esperados. Los métodos anteriores están disponibles con un parámetro adicional de tipo *String: assertEquals (String, Object, Object), assertTrue (String, Boolean).* El parámetro adicional puede usarse para mostrar un mensaje explicativo en caso de error. El método *assertEquals* también está disponible con otras signaturas además de *Object: int, long, float, double*.

Si se espera que el método dispare una excepción con los parámetros reales dados, entonces la etiqueta *@Test* toma como parámetro la excepción esperada.

Incluimos algunos detalles de la clase *TestRacionaJunit*

```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestRacionalJUnit {
    Racional r;

   @Before
   public void setUp() throws Exception {  }

   @Test
   public void testRacionalImplIntegerInteger() {
          r = Racionales.create(4, 6);
          int n = r.getNumerador();
          int d = r.getDenominador();
          assertEquals(n,2);
          assertEquals(d,3);
          r = Racionales.create(-2, -3);
          n = r.getNumerador();
          d = r.getDenominador();
          assertEquals(n,2);
          assertEquals(d,3);
   }

   @Test(expected=IllegalArgumentException.class)
   public void testRacionalImplIntegerInteger1() {
          r = Racionales.create(1, 0);
   }

   @Test
   public void testRacionalImplInteger() {
          r = Racionales.create(3);
          int n = r.getNumerador();
          int d = r.getDenominador();
          assertEquals(n,3);
          assertEquals(d,1);
   }

   @Test
   public void testRacionalImplString() {
          r = Racionales.create("-8/4");
```

```
            int n = r.getNumerador();
            int d = r.getDenominador();
            assertEquals(n,-2);
            assertEquals(d,1);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testRacionalImplString1() {
            r = Racionales.create("2 3");
    }

    @Test
    public void testGetNumerador() {
            r = Racionales.create(-5,9);
            int n = r.getNumerador();
            assertEquals(n,-5);
    }

    @Test
    public void testSetNumerador() {
            r = Racionales.create(4, 9);
            r.setNumerador(5);
            int n = r.getNumerador();
            int d = r.getDenominador();
            assertEquals(n,5);
            assertEquals(d,9);
    }
    ...
}
```

De la misma forma implementaríamos el resto de los casos de prueba. En general esta clase es conveniente implementarla cuando se conocen los detalles del tipo.

9. Diseño de Tipos genéricos

Los **tipos genéricos** requieren algunas consideraciones particulares porque dependen de parámetros y posiblemente estos parámetros pueden tener restricciones. Estas restricciones deben ser expresadas en la especificación Existe la posibilidad de restringir los tipos en que puede instanciarse un parámetro genérico. Esto se consigue introduciendo restricciones en la declaración del parámetro genérico (sea la declaración en interfaces, clases o métodos) o en

los posibles usos de ese parámetro genérico en tipos de los parámetros formales de los métodos genéricos o de los métodos de las clases genéricas.

Las restricciones se expresan con los operadores *?, extends, super, &*. El significado de estos operadores es el siguiente:

- *?* Cualquier tipo
- *T1 extends T2* T1 debe ser un subtipo de T2
- *T1 super T2* T1 debe ser un supertipo de T2
- *R1 & R2* Conjunción de las restricciones R1 y R2

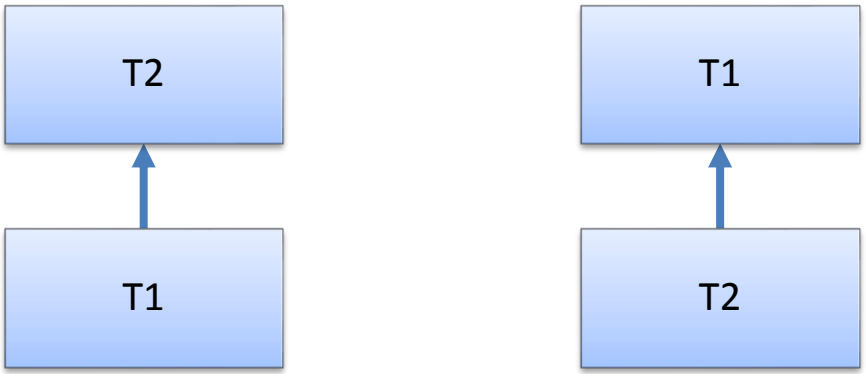


Figura 6. Representación gráfica de los operadores *extends* y *super*

Como hemos visto anteriormente T1 es un subtipo de T2 si T2 es un antecesor de T1 en el grafo de tipos tal como lo definimos en el tema 1. Si T1 es un subtipo de T2 entonces T2 es un supertipo de T1. Recordemos que T1 es un subtipo de T2 si:

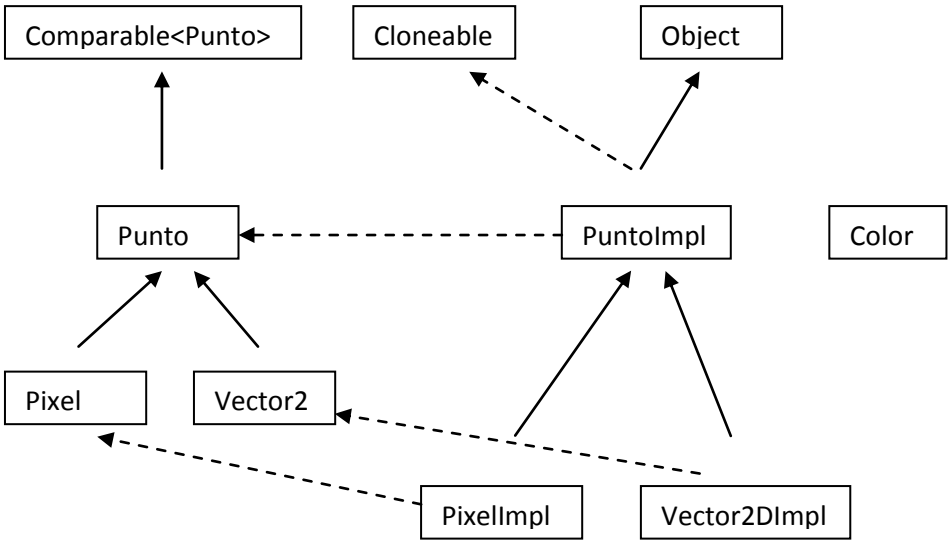
- T1 es una interfaz y extiende directamente o en varios pasos T2 (otra interfaz).
- T1 es una clase, T2 otra clase y T1 hereda de T2 directamente o en varios pasos.
- T1 es una clase, T2 una interfaz y T1 implementa T2 directamente o en varios pasos.
- Todas las clases heredan implícitamente de la clase *Object*.

A partir de lo anterior podemos construir el grafo de tipos siguiendo los siguientes pasos:

- Los nodos del grafo son todos los tipos disponibles
- Cuando un tipo T1 es subtipo directo (en un paso) de otro T2 añadimos un arista de T1 a T2. Esto puede ocurrir si T1 extiende T2 directamente (siendo T1 y T2 ambos interfaces o clases) o cuando T1 es una clase e implementa directamente T2.

- Cuando una clase no extiende explícitamente a otra hay que añadir una arista de ese tipo a *Object*.

En el ejemplo de *Punto, Pixel* del tema 1, suponiendo que *Punto* extiende *Comparable<Punto>* y *PuntoImpl* implementa *Cloneable*, el grafo de tipos es:



Grafo de tipos

Veamos algunos ejemplos de métodos genéricos con restricciones de instanciación:

```
static <T extends Comparable<T>> void sort1(List<T> v);
static <T extends Comparable<? super T>> void sort2(List<T> v);
static <T extends Comparable<? super T> & Cloneable>void sort3(List<T>v);
static <T> void swap1(List<T> v, int i, int j);
static void swap2(List<?> v, int i, int j);
```

En las declaraciones de método anteriores aparecen diversos tipos genéricos, declaraciones de parámetros genéricos y usos de esos parámetros genéricos para declarar tipos de parámetros formales.

En los primeros métodos se declara el parámetro genérico *T*. Como norma general para los parámetros genéricos usaremos una letra única mayúscula: *T, E …*

Comparable<T> es un tipo genérico que se podrá instanciar sustituyendo T por cualquier tipo. *Comparable<? super T>* es otro tipo genérico que se podrá instanciar por cualquier tipo que sea un supertipo (ver grafo de tipos) de *T. List<T>* es un nuevo tipo genérico que se podrá instanciar sustituyendo *T* por cualquier tipo. *List<?>* es un tipo genérico también que se instanciaría sustituyendo el operador *?* por cualquier tipo. La diferencia entre *List<T>* y *List<?>* es que las posibles instanciaciones del primero vienen restringidas por las posibles restricciones que tenga *T* en su declaración. *List<?>* no tiene ninguna restricción para ser instanciado.

Veamos ahora, en los ejemplos anteriores, las restricciones en los parámetros genéricos en la declaración de métodos.

```
static <T extends Comparable<T>> void sort1(List<T> v);
```

La definición del método genérico sort1 declara un parámetro genérico *T* con la restricción que debe extender al tipo *Comparable<T>* instanciado en el mismo tipo *T*. El parámetro formal es de tipo *List<T>* que puede instanciarse al mismo tipo *T* anterior. ¿Qué tipos del grafo de tipos anterior son adecuados para poder instanciar *T*? Podemos comprobar que solo *Punto*. De los métodos *sort1, sort2, sort3* vemos que *sort1* no puede ordenar un *List<Pixel>* porque T no se puede instanciar en *Pixel.*

```
static <T extends Comparable<? super T>> void sort2(List<T> v);
```

En la declaración del método sort2 el parámetro genérico T es como antes, pero ahora con la restricción de que extienda *Comparable<? super T>*. Es decir que extienda Comparable instanciado algún supertipo de *T*. El parámetro formal es de tipo *List<T>* que puede instanciarse al mismo tipo *T* anterior. ¿Qué tipos del grafo de tipos anterior son ahora adecuados para poder instanciar T? Podemos ver que serían válidos *Punto, Pixel, PuntoImpl, PixelImpl*. Pero no con *Object* directamente. Por tanto, el método *sort2 no* puede ordenar un *List< Object >* porque *T* se no puede instanciar en *Object*, pero sí puede ordenar un *List<Pixel >* porque *T* sí se puede instanciar en *Pixel.*

```
static <T extends Comparable<? super T> & Cloneable>void sort3(Vector<T>v);
```

En la declaración del método sort3 el tipo T debe cumplir las restricciones especificadas en el caso dos y además debe extender el tipo *Cloneable,* predefinido en Java*,* que no es genérico. Ahora podemos comprobar que el único tipo que puede instanciar esas restricciones es *PuntoImpl*. Entonces podremos ordenar un *List<PuntoImpl>* pero no un *List<Punto>*, etc.

```
static <T> void swap1(List<T> v, int i, int j);
static void swap2(List<?> v, int i, int j);
```

En el método swap1 se declara el parámetro genérico *T* sin restricciones en la declaración. Tampoco hay restricciones en el parámetro formal, por tanto, *T* puede ser instanciado a cualquier tipo. La declaración del método swap2 es equivalente a la de swap1 pero más simple por no necesitar declarar ningún parámetro genérico.

Como ejemplo veamos el tipo intervalo cerrado de un tipo con orden natural.

Intervalo sobre un tipo genérico *T* que tenga un orden natural.

Intervalo de T, T tiene que tener orden natural
Descripción: Un intervalo de valores de tipo T

- *Propiedades:*
  - *LímiteInferior*, tipo T, solo consultable
  - *LímiteSuperior*, tipo T, solo consultable, debe ser mayor que el límite inferior según el orden natural de T
  - *esInterior*, *boolean*, decide si un objeto de tipo T está dentro del intervalo incluidos los bordes
  - *estáIncluido, boolean*, decide si un intervalo dado está incluido en *this*
- *Otras propiedades*:
  - *Criterio de igualdad*: deben iguales el límite inferior y el superior
  - *Orden natural*: no tiene
  - Representación como cadena: límite inferior, superior, separados por comas y entre corchetes

El diseño del tipo es:

```
public class Intervalo<T extends Comparable<? super T>>  {
    T getLimiteInferior() {...}
    T getLimiteSuperior() {...}
    Boolean esInterior(T e) {...}
    Boolean estaIncluido(Intervalo<T> i) {...}
    public static <T extends Comparable<? super T>> Intervalo<T>
              create(T li, T ls) { ...}
}
```

En los casos de prueba representamos por *[a,b]* un intervalo con límite superior b e inferior a.

Casos de prueba para el tipo Intervalo<T> instanciado con el tipo *Racional.*

Método	this	p	r	Excepción
Constructores	[(3,7),(1,7)]			IllegalArgumentException
getLimiteInferior	[(2,4),(5,6)]		(1,2)	
getLimiteSuperior	[(2,4),(5,6)]		(5,6)	
esInterior	[(2,4),(5,6)]	(1000,5)	false	
toString	[(2,6),(-8,-4)]		"[1/3,2]"	
...				

10. **Otros problemas de diseño**

En muchos casos es necesario dotar a varios tipos de un conjunto de operaciones comunes. Es el caso de los objetos geométricos en dos dimensiones. Entre estas operaciones comunes podemos incluir *rotar*, *trasladar* y *mostrar (draw*) en la pantalla.

Este objetivo puede conseguirse diseñando un interface con los métodos adecuados y haciendo que cada uno de los objetos geométricos lo implementen. Una posibilidad es:

```
public interface ObjetoGeometrico2D {
       ObjetoGeometrico2D rota(Punto2D p, Double angulo);
       ObjetoGeometrico2D traslada(Vector2D v);
       void draw(Graphics2D g);
}
```

Los tipos *Punto2D, Recta2D, Semiplano2D, Poligono2D* deberían implementar este interface. También podemos diseñar un agregado de objetos geométricos que nos permita aplicar las operaciones anteriores de forma conjunta. La implementación de este agregado se muestra abajo.

Tipo ObjetoGeometricoAgregado2D implementa ObjetoGeométrico2D

Descripción: Un agregado de objetos geométricos

Propiedades:

- *ObjetosGeometricos:* `List<ObjetoGeometrico2D>`

Operaciones:

- *rota(Punto2D p, Double angulo):* ObjetoGeometrico2D. Construye un nuevo objeto agregado con los objetos geométricos rotados.
- *traslada(Vector2D v):* ObjetoGeometrico2D. Construye un nuevo agregado con los objeto geométrico rotados.
- *draw(Graphics2D g): void.* Representa el objeto geométrico
- *add(ObjetoGeometrico2D a): void.* Añade un objeto geométrico al agregado.

Factoría

- *ObjetoGeometricoAgregado2D create() { … }*
- *ObjetoGeometricoAgregado2D create(List<ObjetoGeometrico2D> objetosGeometricos) { … }*
- *ObjetoGeometricoAgregado2D create(ObjetoGeometricoAgregado2D a) { … }*

11. Casos de prueba para las propiedades derivadas del tipo Object y Comparable

Las propiedades que se heredan de *Object (equals, hashCode y toString*) y otras propiedades generales como *Comparable* tienen unos requisitos generales y unas restricciones entre ellas que deben ser comprobados en cada implementación. Los requisitos de cada propiedad y las restricciones entre ellas fueron explicados anteriormente. Para comprobar cada una de estas

propiedades implementamos la clase *TestObject.* Cada método de esta clase comprueba una de las propiedades. Así por ejemplo la propiedad reflexiva de la igualdad es de la forma:

$$\forall\, T\ o\colon o.equals(o)$$

Es decir para todo objeto *o* de un tipo *T*, en este caso cualquier tipo, siempre es verdad que *o* es igual a sí mismo. Otra propiedad es la transitiva de la igualdad

$$\forall\, T\ o1, o2, o3\colon o1.equals(o2)\ \&\&\ o2.equals(o3)\ \rightarrow\ o1.equals(o3)$$

Pretendemos comprobar que las propiedades se cumplen para los casos de prueba que diseñemos. El código que comprueba la propiedad se muestra abajo.

En este caso concreto se comprueban las propiedades generales derivadas de *Object* para el tipo *Fecha.*

```
public class TestObjectTest2 {
    Fecha f1;
    Fecha f2;
    Fecha f3;
    Fecha f4;
    @Before
    public void setUp() throws Exception {
           f1 = Fechas.create(1, 1, 2000);
           f2 = Fechas.create("1 de Enero de 2000");
           f3 = Fechas.create(1, 1, 2000);
           f4 = Fechas.create(20, 2, 2009);
    }

    @Test
    public void testReflexivaIgualdad() {
           assertTrue(f1.equals(f1));
    }

    @Test
    public void testSimetricaIgualdad() {
           assertTrue(f1.equals(f2)==f2.equals(f1));
    }

    @Test
    public void testTransitivaIgualdad() {
```

```
            assertTrue(!(f1.equals(f2)&& f2.equals(f3)) ? true :
                f1.equals(f3));
    }

    @Test
    public void testIgualdadHashCode() {
            assertTrue(!(f1.equals(f2)) ? true :
                f1.hashCode()==f2.hashCode());
    }

    @Test
    public void testIgualdadToString() {
            assertTrue(!(f1.equals(f2)) ? true :
                f1.toString().equals(f2.toString()));
    }

    @Test
    public void testCopiable() {
            Fecha f = f1.clone();
            assertTrue(f.equals(f1) && f!=f1);
    }

    @Test
    public void testComparableIgualdad() {
            assertTrue((f1.equals(f2) == (f1.compareTo(f2) == 0)));
            assertTrue((f1.equals(f4) == (f1.compareTo(f4) == 0)));
    }

    @Test
    public void testComparableAntisimetrica() {
            assertTrue(Utiles.sgn(f1.compareTo(f4)) ==
                        -Utiles.sgn(f4.compareTo(f1)));
    }

    @Test
    public void testComparableTransitiva() {
            assertTrue(!(f1.compareTo(f2)<=0 && f2.compareTo(f3)<=0 ) ?
                        true : f1.compareTo(f3)<=0);
    }
    @Test
    public void testComparableTransitiva2() {
            assertTrue(!(f1.compareTo(f2)<=0 && f2.compareTo(f4)<=0 ) ?
                        true : f1.compareTo(f4)<=0);
    }
}
```

12. Conceptos aprendidos

- Métodos de la clase *Object: equals, toString, hashCode*
- Comparación de objetos por su orden natural: el tipo Comparable
- Comparación de objetos por órdenes alternativos: el tipo *Comparator*
- Copia de objetos: clone y copia en profundidad
- Procedimiento para el diseño de tipos
- Especificación de casos de prueba para los tipos
- Diseño y prueba de tipos genéricos
- Creación de expresiones con objetos: el tipo Expresión
- Evaluación de condiciones sobre objetos: el tipo Criterio
- Cambio de las propiedades de objetos: el tipo Acción
- Combinación de expresiones, criterios y acciones
- Ejecución de los casos de prueba: uso de la herramienta jUnit

13. Ejercicios Propuestos

Diseñar los tipos siguientes. En cada caso se trata de definir las propiedades (de las que se da algunas y se pueden incluir otras), las operaciones, las propiedades de la igualdad, la factoría y los casos de prueba.

Las diversas propuestas están presentadas con diversos grados de detalle. Cada una de ellas sólo indica posibilidades para el tipo. Se trata de concretarlas, modificarlas si es necesario y aclarar los aspectos de diseño.

Los problemas de implementación se abordarán en el capítulo siguiente.

1. **Tupla2 de T1 y T2**

Descripción: Objetos que representan pares de valores: es decir el producto cartesiano de los tipos *T1* y *T2*. Inmutable

Propiedades:

- *P1: T1*, consultable.
- *P2: T2*, consultable

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen las mismas propiedades.
- *Representación como cadena*: valor de las propiedades separadas por comas y entre paréntesis.

Factoría:

- *Tupla2<T1,T2> create(T1 p1, T2, p2):* Crea una tupla con las propiedades p1 y p2.
- *Tupla2<T1,T2> create(Tupla2<T1,T2> t):* Crea una copia de t
- *Tupla2<T,T> create(List<T> lista):* Crea una tupla cuyos valores son los dos primeros valores de una lista de dos elementos.
- *Tupla2<T1,T2> create(T[] a):* Crea una tupla cuyos valores son los dos primeros valores de un array de dos elementos.

2. **Intervalo de T**

Descripción: Objetos que representan intervalos de valores de tipo T. Inmutable

Propiedades:

- *LímiteInferior*: *T,* consultable.
- *LimiteSuperior*: *T*, consultable
- *Contiene*(T elem): *Boolean,* Consultable. Si contiene al elemento indicado.
- *Contiene(Intervalo<T> inter): Boolean,* Consultable. Si contiene al intervalo indicado.

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen el mismo límite inferior y superior.
- *Representación como cadena*: valor de los límites inferior y superior separados por comas y entre paréntesis.

Factoría:

- *`Intervalo<T> create(T li, T ls):`* Crea un intervalo dados los límites. El límite inferior debe ser menor o igual que el superior.

3. ObjetoGeometrico2D

Descripción: Objetos que pueden ser representados en el plano.

Operaciones:

- *`ObjetoGeometrico2D rota(Punto2D p, Double a):`* `Un nuevo objeto rotado el ángulo` *`a`* `con respecto al punto` *`p`*`.`
- *`ObjetoGeometrico2D traslada(Vector2D v):`* `Un nuevo objeto trasladado según el vector v.`

4. ObjetoGeometricoAgregado2D extiende ObjetoGeometrico2D

Descripción: Un agregado de objetos geométricos em *2D*. Mutable.

Propiedades:

- *ObjetosGeometricos*: *List<ObjetoGeometrico2D>*, consultable.

Propiedades relacionadas con la igualdad:

- *Criterio de igualda*d: dos objetos de este tipo son iguales si tienen los mismos objetos geométricos.
- *Representación como cadena*: Secuencia de objetos geométricos separados por comas.

Operaciones:

- *void add(ObjetoGeometrico2D a):* Añade un nuevo objeto geométrico

5. Punto2D extiende ObjetoGeometrico2D

Descripción: Puntos en el plano. Mutable

Propiedades:

- *X: Double*, consultable, modificable.
- *Y: Double*, consultable, modificable
- *Origen: Punto2D,* consultable, compartida
- *DistanciaA(Punto2D p): Double,* consultable. Distancia al punto p.
- *DistanciaAlOrigen: Double,* consultable. Distancia al origen

Propiedades relacionadas con la igualdad

- *Criterio de igualdad*: dos puntos son iguales si tienen la misma X e Y.
- *Representación como cadena*: valor de las propiedades X e Y separadas por comas y entre paréntesis.
- *Orden Natural*: Primero según las X. Luego según las Y.

Operaciones:

- *Punto2D add(Vector2D v):* Construye el punto resultante de sumar el vector v a this.
- *Vector2D minus(Punto2D v):* Construye el vector resultante de restar el vector v de *this.*

Factoría:

- *Punto2D create(Double x, Double y):* Crea un punto a partir de sus propiedades
- *Punto2D create(Punto2D p):* Crea una copia del punto
- *Punto2D create():* Crea el punto (0.0.)
- *Punto2D create(Vector2D v):* Crea un punto a partir de un vector

6. Vector2D

Descripción: Vector en el plano. Mutable

Propiedades:

- *X: Double*, consultable, modificable.

- *Y: Double*, consultable, modificable
- *Angulo: Double,* consultable, modificable. Angulo en radianes
- *Modulo: Double,* consultable, modificable
- *AnguloEnGrados*: Double, consultable.
- *Ortogonal: Vector2D,* consultable. Vector con el mismo módulo y girado 90 grados
- *Unitario: Vector2D,* consultable. Vector con el mismo ángulo y módulo 1.
- *Opuesto: Vector2D,* consultable.

Invariante:

- $Modulo = \sqrt{X^2 + Y^2}$
- $Angulo = \tan^{-1}\frac{Y}{X}$

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen las mismas propiedades.
- *Representación como cadena*: valor de las propiedades separados por comas y entre paréntesis.

Operaciones:

- *Vector2D proyectaSobre(Vector2D v):* Observadora. Devuelve le producto escalar de *this* por el vector unitario em la dirección de v.
- *Vector2D suma(Vector2D v):* Observadora.
- *Vector2D multiplica(Double factor):*Observadora. Devuelve *(factor*X, factor*Y).*
- *Double multiplicaVectorial(Vector2D v):* Observadora. Devuelve this.*X*v.Y-this.Y*v.X*
- *Double multiplicaEscalar(Vector2D v):* Observadora. Devuelve this.*X*v.X+this.Y*v.Y*
- *Vector2D rota(Double angulo).* Observadora

Factoría:

- `Vector2D createCartesiano(Double x, Double y)`
- `Vector2D create(Punto2D p)`
- `Vector2D create(Vector2D p)`
- `Vector2D createPolarEnGrados(Double modulo, Double angulo)`
- `Vector2D createPolarEnRadianes(Double modulo, Double angulo)`

7. Recta2D extends ObjetoGeometrico2D

Descripción: Recta en el plano. Inmutable. Ecuación de la recta $Ax + By + C = 0$.

Propiedades:

- *A: Double*, consultable.
- *B: Double*, consultable.
- *C: Double,* consultable.
- *Punto: Punto2D*, consultable.
- *Vector: Vector2D, consultable.*
- *Angulo: Double,* consultable. Angulo en radianes
- *AnguloEnGrados*: Double, consultable.
- *DistanciaAlOrigen*: Double, consultable.
- *Distancia(Punto2D p)*: Double, consultable.
- *Paralela(Punto2D p)*:
- *Perpendicular(Punto2D p)*:
- *Semiplano*: Semiplano2D, consultable.

Invariante:

- $Vector = (B, -A)$
- $Distancia(p) = \frac{Ax+By+C}{\sqrt{A^2+B^2}}$

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen la misma distancia al origen (teniendo en cuenta el signo) y la diferencia de sus ángulos es 0 o 180.
- *Representación como cadena*: de la forma $Ax + By + C = 0$

Operaciones:

- *boolean contienePunto(Punto2D p):* Observadora. Si $Ax + By + C = 0$
- *Punto2D cortaA(Recta2D v):* Observadora.

Factoría:

- *Recta2D createEnRadianes(Punto2D p, Double angulo).*

- *Recta2D createEnGrados(Punto2D p, Double angulo).*
- *Recta2D create(Punto2D p, Double angulo).*
- *Recta2D create(Punto2D p, Vector2D d).*
- *Recta2D create(Punto2D p1, Punto2D p2).*
- *Recta2D create().* Devuelve el eje X.

8. Semiplano2D extends Recta2D

Descripción: Semiplano en 2D. Inmutable

Propiedades:

- *Opuesto: Semiplano2D*, consultable.

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen la misma distancia al origen (teniendo en cuenta el signo) y sus ángulos iguales.
- *Representación como cadena*: de la forma $Ax + By + C < 0$

Operaciones:

- *boolean contiene(Punto2D p):* Observadora. Si $Ax + By + C < 0$
- *boolean contiene(Poligono2D p):* Observadora.
- *Poligono2D intersecta(Poligono2D v):* Observadora.
- *Recta2D asRecta2D().*

Factoría:

- *Semiplano2D create(Punto2D p, Vector2D d);*
- *Semiplano2D create(Recta2D r);*
- *Semiplano2D create(Semiplano2D r);*

9. **Polígono2D extends ObjetoGeometrico2D**

Propiedades:

- *Vértices*: *List<Punto>,* consultable.
- *NúmeroDeVértices>*: Entero, consultable.

- *Lados*: *List<Vector2D>,* consultable.
- *Perímetro*: *Double*, consultable.
- *Area*: *Double*, consultable

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos polígonos son iguales si tienen los mismos vértices.
- *Representación como cadena*: secuencia de vértices separados por comas y entre llaves.

Operaciones

- *añadeVertice*: añade un vértice (de tipo Punto) al polígono.

Factoría:

- *Poligono2D create().*
- *Poligono2D create(Punto2D p1, Punto2D p2, Punto2D p3).*
- *Poligono2D create(Punto2D... lp).*
- *Poligono2D createRectangulo(Punto2D p, Double base, Double altura).*
- *Poligono2D createRectangulo(Double xMin, Double xMax, Double yMin, Double yMax).*

10. Racional

Descripción: Racional. Inmutable

Propiedades:

- *Numerador: Integer*, consultable.
- *Denominador: Integer*, consultable
- *ValorReal: Double,* consultable.

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos racionales $\frac{a}{b}, \frac{c}{d}$ de este tipo son iguales si $ad = bc$
- *Representación como cadena*: de la forma a/b si b es distinto de 1 en otro caso a.
- *Orden natural*: el usual en los racionales

Operaciones:

- *Racional suma(Racional r).*
- *Racional resta(Racional r)*
- *Racional multiplica(Racional r)*
- *Racional divide(Racional r)*
- *Racional invierte().*

Factoría:

- *Racional getCero().*
- *Racional getUno().*
- *Racional create(Integer a, Integer b).*
- *Racional create(Integer a).*
- *Racional create(String s).*
- *Racional create(Racional r).*

11. Complejo

Descripción: Complejo. Inmutable

Propiedades:

- *ParteReal*, Double, consultable.
- *ParteImaginaria*, Double consultable.
- *Modulo*, Double, consultable
- *Argumento, Double,* consultable

Propiedades de la igualdad:

- Criterio de igualdad: dos complejos son iguales si tienen iguales su parte real e imaginaria.
- Representación como cadena: la parte real seguida de la imaginaria terminada en *i*. Si la parte real o imaginaria son cero no aparecerán. El complejo con ambas partes nulas se representará como 0.

Operaciones:

- *Complejo suma(Complejo r).*

- *Complejo resta(Complejo r)*
- *Complejo multiplica(Complejo r)*
- *Complejo divide(Complejo r)*
- *Complejo conjugado()*.

Factoría:

- *Complejo getCero().*
- *Complejo getUno().*
- *Complejo create(Double a, Double b).*
- *Complejo create(Double a).*
- *Complejo create(String s).*
- *Complejo create(Complejo r).*

12. Polinomio de Reales

Descripción: Polinomio de Reales. Inmutable

Propiedades:

- *Grado*, de tipo *Integer*, consultable.
- *Coeficiente(Integer indice)*, de tipo *Double*, consultable. Devuelve el coeficiente del monomio de grado indice.
- *Coeficientes*, *List<Double>,* consultable.
- *Valor(Double x).* Valor del polinomio.

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos polinomios son iguales si tienen iguales su y grado y coeficientes.
- *Representación como cadena*: una secuencia de monomios, donde cada monomio es representado por el coeficiente, con su signo, seguido del literal “X”, el símbolo “^” y el grado correspondiente.
- *Orden Natural*: por grado y si es igual por el coeficiente de monomio de mayor grado.

Operaciones:

- *Polinomio add(Polinomio p)*
- *Polinomio negate()*
- *Polinomio multiply(Double e)*
- *Polinomio multiply(Polinomio p).*
- *Polinomio subtract(Polinomio p)*
- *Polinomio derivative().*

Factoría:

- *Polinomio create().*
- *Polinomio create(Double[] coeficientes).*
- *Polinomio create(Double coef, int grado).*
- *Polinomio create(Polinomio p)*
- *Polinomio getZero().*
- *Polinomio getOne()*

13. Fecha

Descripción: Fechas del calendario gregoriano. Inmutable

Propiedades:

- *Dia*, entero entre 1 y, dependiendo del mes, 30, 31, 28 ó 29, consultable.
- *DiaSemana*, *String*, solo consultable, derivada
- *Mes*, entero entre 1 y 12 inclusive, solo consultable
- *MesCadena*, *String*, derivada
- *Año*, entero mayor o igual a 1900, consultable
- *Bisiesto(), Boolean. Si el año es bisiesto*

Propiedades de la igualdad:

- *Orden natural*: el usual entre fechas
- *Criterio de Igualdad*: si tienen el mismo día, mes y año
- Representación como cadena: DiaSemana, " a ", Día, " de ", MesCadena, " de ", Año

Operaciones:

- *Fecha suma(Integer a).* Devuelve una fecha posterior en el número de días indicado por el parámetro.
- *Fecha resta(Integer a).* Devuelve una fecha anterior en el número de días indicado por el parámetro
- *Integer resta(Fecha f).* Devuelve un entero, positivo, negativo o cero, con el número de días transcurridos entre las fechas representadas por el objeto y el parámetro

Factoría:

- *Fecha create(Integer d, Integer m, Integer a).*
- *Fecha create(String s).*

14. Para describir una biblioteca de fotos digitales se definen los siguientes tipos:

Foto

Propiedades:

- *Título*, de tipo String, consultable.
- *Autor*, de tipo Persona, consultable.
- *Fecha*, de tipo Fecha, consultable.
- *Formato*, de tipo String, consultable.
- *Alto*, de tipo Integer, consultable.
- *Ancho*, de tipo Integer, consultable.
- *Tamaño*, de tipo Double, consultable.
- *Descriptores*, de tipo List<String>, consultable.
- *Calificación*, de tipo Integer, consultable y modificable.
- *Retocada*, de tipo Boolean, consultable y modificable.
- *NúmeroDeVistas*, de tipo Integer, consultable y modificable.

Propiedades de la igualdad:

- Criterio de igualdad: dos fotos son iguales si tienen el mismo título y el mismo autor.

- Representación como cadena: [Título, autor]
- Orden natural: por título, y a igualdad de título por autor.

Álbum

Propiedades:

- *Nombre*, de tipo String, consultable.
- *Año*, de tipo Integer, consultable.
- *Fotos*, de tipo List<Foto>, consultable y modificable.
- *NúmeroDeFotos*, de tipo Integer, consultable (propiedad calculada a partir del vector de fotos)
- *Tamaño*, de tipo Double, consultable (propiedad calculada a partir del vector de fotos)

Propiedades de la igualdad:

- Criterio de igualdad: dos álbumes son iguales si tienen el mismo nombre y el mismo año.
- Representación como cadena: [Nombre, año]

Biblioteca

Propiedades:

- *Propietario*, de tipo String, consultable y modificable.
- *Álbumes*, de tipo List <Álbum>, consultable y modificable.
- *NúmeroDeÁlbumes*, de tipo Integer, consultable (propiedad calculada a partir del vector de álbumes)

Propiedades de la igualdad:

- Criterio de igualdad: dos bibliotecas son iguales si tienen el mismo propietario.
- Representación como cadena: [Propietario]

Persona.

Propiedades:

- *Nombre*, de tipo String, consultable y modificable.
- *Apellidos*, de tipo String, consultable y modificable.
- *Dni,* de tipo String, consultable.
- *Edad*, de tipo Integer, consultable y modificable.

Propiedades de la igualdad:

- Criterio de igualdad: por Dni, apellidos y nombre.
- Representación como cadena: [Apellidos, Nombre, Dni]

15. El tío Hortensio utiliza la siguiente relación de tipos para gestionar sus huertos. Cada uno de sus huertos está organizado en diferentes cosechas, según el tipo de producto que cultiva en ellas.

Cosecha. Comparable

Propiedades:

- *Beneficio*, de tipo Double, consultable y modificable.
- *TiempoRecolección* (en días), de tipo Integer, consultable y modificable.
- *TiempoPutrefacción* (días que tarda en pudrirse la cosecha), de tipo Integer, consultable y modificable. Siempre será mayor o igual que TiempoRecolección.
- *NombreProducto*, de tipo String, consultable.
- *Extensión* (en m2), de tipo Double, consultable y modificable

Propiedades de la igualdad:

- Criterio de igualdad: por beneficio y nombre de producto.
- Orden natural: por beneficio y a igualdad de beneficio por nombre de producto.
- Representación como cadena: [Nombre del producto: beneficio]

Huerto.

Propiedades:

- *Cosechas*, de tipo List <Cosecha>, consultable.
- *Clientes*, de tipo List <Persona>, consultable.
- *Nombre*, de tipo String, consultable.
- *Localización*, de tipo String, consultable y modificable.

Propiedades de la igualdad:

- Criterio de igualdad: Dos huertos serán iguales si tienen el mismo nombre y las mismas cosechas.
- Representación como cadena: [Nombre: vector de cosechas]

Operaciones:

- *añadeCosecha*, permite añadir una cosecha al huerto.
- *añadeCliente*, permite añadir un nuevo cliente.

16. Se desea modelar un dispositivo electrónico que permite almacenar libros, junto con otros tipos de documentos en formato digital como periódicos, revistas o artículos. Para ello se definen los siguientes tipos de datos:

Escritor. Comparable

Propiedades:

- *Nombre*, de tipo String, consultable
- *Apellidos*, de tipo String, consultable
- *Nacionalidad*, de tipo String, consultable y modificable

Propiedades de la igualdad:

- Orden natural: un Escritor es mayor que otro si su apellido es mayor, en caso de igualdad si su nombre es mayor, y en caso de igualdad si su nacionalidad es mayor
- Criterio de igualdad: dos escritores son iguales si tienen el mismo nombre, apellidos y nacionalidad
- Representación como cadena: apellidos seguidos de una coma y el nombre y entre paréntesis la nacionalidad

Libro.

Propiedades:

- *Título*, de tipo String, consultable
- *Autor*, de tipo Escritor, consultable
- *Año de publicación*, de tipo entero, mayor o igual que 1000, consultable
- *Idioma*, de tipo String, consultable
- *Precio*, de tipo real, mayor o igual que cero, consultable y modificable
- *Valoración*, de tipo Integer, entre 0 y 5, consultable y modificable
- *Descriptores*, de tipo List <String>, consultable y modificable

Propiedades de la igualdad:

- Orden natural: un Libro es mayor que otro si su título es mayor, en caso de igualdad si su autor es mayor, y en caso de igualdad si su año de publicación es mayor
- Criterio de igualdad: dos libros son iguales si tienen el mismo título, autor y año de publicación
- Representación como cadena: título seguido de una coma, el autor, otra coma y el año de publicación

LectorElectrónico de T

Propiedades:

- *Nombre*, de tipo String, consultable y modificable
- *Capacidad*, de tipo real, consultable
- *Documentos*, de tipo List <T>, que representa los documentos almacenados en el lector, consultable y modificable

17. Se desea realizar una aplicación para gestionar información relativa a los viajes vendidos en una agencia de viajes. Para ello se definen los siguientes tipos:

    Operador.

    Propiedades:

- *Nombre*, de tipo String, consultable.
- *CIF*, de tipo String, consultable.
- *Dirección*, de tipo String, consultable y modificable.

Propiedades de la igualdad:

- Criterio de igualdad: Dos operadores son iguales si tienen el mismo nombre y el mismo CIF.
- Representación como cadena: [nombre, CIF].

Viaje. Comparable.

Propiedades:

- *Identificador*, de tipo String, consultable.
- *CompañiaOperadora*, de tipo Operador, consultable.
- *MínimoDePersonas*, de tipo Integer, consultable.
- *CiudadesVisitadas*, de tipo List <String>, consultable y modificable;
- *Precio*, de tipo Double, consultable y modificable. Debe tomar un valor mayor que cero.
- *Calificacion*, de tipo Integer, consultable y modificable. Debe tomar un valor mayor o igual que cero y menor o igual que diez.

Propiedades de la igualdad:

- Orden natural: Por identificador, y a igualdad de éste, por precio.
- Criterio de igualdad: Dos viajes son iguales si tienen el mismo identificador y el mismo precio.
- Representación como cadena: [identificador, precio].

Donde:

Un operador tiene un nombre, un CIF o número de identificación fiscal y una dirección. Un viaje tiene un identificador, una compañía operadora o empresa encargada de realizar el viaje, un número mínimo de personas para que el viaje se pueda realizar, las ciudades que

se visitan, su precio y una calificación, entre 0 y 10, que indica el grado de satisfacción de los clientes con el viaje.