# Diseño de software con objetos - estudio de casos

Autor: Carlos Lombardi

El propósito de este documento es presentar algunas ideas sencillas que pueden ayudar a la hora de plantear un modelo de objetos, a partir de un caso de aplicación de cada una.
Para cada caso, se describe la solución y se muestra una implementación en Java 8.

Varios de los casos son tomados de las guías compiladas por Carlos Lombardi y Leonardo Gassman para la materia Programación con Objetos 1, Universidad Nacional de Quilmes. Estas guías se pueden acceder desde https://objetos1wollokunq.gitlab.io/material/ . Nos referiremos a este material como PO1UNQ.

# Distribuir la resolución

En el ejercicio 2 guía 10 PO1UNQ, se pide modelar algunos aspectos del funcionamiento de un conjunto de empresas. De cada empresa se nos dice que está organizada en **centros de distribución**. A su vez, cada centro de distribución maneja varios **vendedores**.
El enunciado también define *certificaciones* que pueden obtener los vendedores, cada una con un puntaje. De cada vendedor se sabe qué certificaciones obtuvo.
Asimismo, se indica qué condiciones debe cumplir un vendedor para *poder trabajar* en una ciudad, esto depende del tipo de vendedor. Se definen tres tipos de vendedores: vendedores fijos, viajantes, y comercios corresponsales. Cada uno define una condición distinta para saber si puede o no trabajar en una ciudad.

Transcribimos el enunciado de dos de los puntos

**Punto 3**

> Para una empresa, obtener el conjunto formado por el *vendedor más flojo de cada centro* de distribución. El vendedor más flojo de un centro es, de entre los que maneja el centro, el que tiene menor puntaje total de certificaciones.

**Punto 4**

> Saber cuáles de los centros de una empresa pueden *cubrir una ciudad*, para esto alcanza que un vendedor que maneja el centro pueda trabajar en la ciudad.

## Arrancamos por el requerimiento

Estos dos puntos corresponden a dos consultas, que debemos lograr que se le puedan hacer a alguno de los objetos del modelo que estamos construyendo.
Antes de empezar a pensar la resolución, definamos en cada caso,

- a qué objeto le vamos a hacer la consulta,
- enviándole qué mensaje,
- con qué parámetros (si hacen falta parámetros), y
- qué forma debería tener (es decir, de qué **tipo** conviene que sea) el resultado de la consulta, o sea, lo que devuelva el mensaje que decidamos.

En este caso, para ambos puntos elegimos hacerle las consultas a un objeto que represente a la empresa.
Para el punto 3, hay que devolver un *conjunto de vendedores*, uno por cada centro. En este caso no hacen falta parámetros.
Para el punto 4, lo que se espera es un *conjunto de centros*, que es un subconjunto de los centros de la empresa. Acá sí tenemos un parámetro, que es la ciudad.

Hasta acá, llegamos a que vamos a tener algo así en la clase Empresa:

```
public class Empresa {
    public Set<CentroDeDistribucion> centros;

    public Set<CentroDeDistribucion> getCentrosDeDistribucion() {
       return this.centros;
    }

    /**
     * Consulta para el punto 3.
     */
    public Set<Vendedor> getVendedoresMasFlojos() {
       // TODO implementar
    }

    /**
     * Consulta para el punto 4.
     */
    public Set<CentroDeDistribucion> getCentrosQuePuedenCubrir(Ciudad ciudad) {
       // TODO implementar
    }
}
```

## Detectar responsabilidades, asignarlas a objetos

Al pensar cómo resolver el punto 3, puede aparecer una idea de este estilo

> "Recorro los centros, de cada centro me fijo todos los vendedores, de cada vendedor sumo el puntaje de todas las certificaciones que tenga. De cada centro, me quedo con el vendedor que la suma me dé el valor más chico"

Parecen muchas cosas para pensa.rlas todas juntas, también para resolverlas todas en el mismo método. Garpa aplicar la idea de *división de tareas*. A su vez, los distintos objetos que tenemos definidos nos dan un criterio para organizar esta división.

Repasemos lo que dice el enunciado:

> Para una empresa, obtener el conjunto formado por el *vendedor más flojo de cada centro* de distribución.

Ah, acá ya tenemos un criterio de organizacion: pensar en el *vendedor más flojo de cada centro* de distribución. ¿A qué objeto podría ser cómodo pedirle el vendedor más flojo de un centro? Claro, al centro.

Dicho en términos de la "teoría de objetos", detectamos una **responsabilidad** que no teníamos en foco hasta ahora: obtener el vendedor más flojo de un centro.
Para cada responsabilidad hay que definir lo mismo que para un punto: a qué objeto le corresponde la consulta u orden, con qué mensaje, con qué parámetro/s (o si es sin parámetros), qué devuelve (o sea, el tipo de lo que devuelve).

Una vez que me di cuenta de esto, el método en empresa es un map del vendedor más flojo de cada centro, que se lo pido al centro. En código:

```
public Set<Vendedor> getVendedoresMasFlojos() {
	return this.getCentrosDeDistribucion().stream()
			.map(centro -> centro.getVendedorMasFlojo())
			.collect(Collectors.toSet());
}
```

Nos queda resolver el método `getVendedorMasFlojo()` en la clase `CentroDeDistribucion`. Otra vez, nos va a ayudar el enunciado.

> El vendedor más flojo de un centro es, de entre los que maneja el centro, el que tiene menor puntaje total de certificaciones.

Aunque no está tan-tan explícito, esta frase también describe una nueva responsabilidad: la de conocer el *puntaje total de certificaciones de un vendedor.* Otra vez, esto es una consulta. ¿A qué objeto le hacemos la consulta? Claro, al vendedor, al que le vamos a agregar un método `getPuntajeTotalCertificaciones()`.
En la clase `CentroDeDistribucion` nos queda

```
public Vendedor getVendedorMasFlojo() {
    return this.getVendedores().stream()
          .min((vend1, vend2) -> Integer.compare(
              vend1.getPuntajeTotalCertificaciones(),
              vend2.getPuntajeTotalCertificaciones()
          ))
          .get();
}
```

El método en Vendedor es sencillo: un sum de los puntajes de cada certificación:

```
public int getPuntajeTotalCertificaciones() {
    return this.getCertificaciones().stream()
          .mapToInt(cert -> cert.getPuntaje()).sum();
}
```

## Conclusión hasta acá

Al trabajar con objetos, aplicar la idea de división de tareas tiene todo el sentido. En particular, para cada subtarea sirve pensar a qué objeto se la asignamos.
Para detectar qué subtareas nos conviene definir, conviene pensar en la idea de responsabilidad: algo que el programa que estamos armando tiene que hacer, que va a ser resuelto como una consulta u orden a un objeto. Para cada responsabilidad, hay que definir “las cuatro cosas”

- qué objeto va a recibir la consulta u orden,
- con qué mensaje,
- con qué parámetros (si hacen falta parámetros), y
- si es una consulta, qué forma debería tener (es decir, de qué **tipo** conviene que sea) el resultado, o sea, lo que devuelva el mensaje que decidamos.

## Lo mismo con el punto 4

Eso, resolvamos el punto 4 siguiendo las mismas ideas.
Otra vez, el enunciado nos da una pista para organizar las tareas:

> … cuáles de los centros … pueden *cubrir una ciudad*, para esto alcanza que un vendedor que maneja el centro pueda trabajar en la ciudad

Surge una responsabilidad cantada: saber si un centrro de distribución puede o no cubrir una ciudad. Repasamos "las cuatro cosas". Esto **(1)** se lo preguntamos al centro, usando **(2)** el mensaje `puedeCubrirCiudad`, que lleva **(3)** un parámetro que es la ciudad, y **(4)** devuelve un booleano.

Con esto en claro, volvamos al enunciado. Lo que se pide es el subconjunto de los centros de la empresa que cumplen una condición. Esto es un `filter` sobre los centros de distribución de la empresa, usando la pregunta que recién le asignamos al centro.
En `Empresa` queda

```
public Set<CentroDeDistribucion> getCentrosQuePuedenCubrir(Ciudad ciudad) {
  return this.getCentrosDeDistribucion()
     .stream()
     .filter(centro -> centro.puedeCubrirCiudad(ciudad))
     .collect(Collectors.toSet());
}
```

El método `puedeCubrirCiudad(Ciudad)` en CentroDeDistribucion es un `anyMatch`: debe devolver `true` si hay algún vendedor que pueda trabajar en la ciudad.
Encontramos una nueva responsabilidad: saber si un vendedor puede o no trabajar en una ciudad[1].
Esta es una consulta, ¿a qué objeto le preguntamos? A priori hay dos opciones sencillas[2]:

- preguntarle a la ciudad, con el vendedor como parámetro:
  `ciudad.puedeTrabajar(vendedor)`
- preguntarle al vendedor, con la ciudad como parámetro:
  `vendedor.puedeTrabajarEn(ciudad)`

El enunciado nos ayuda a decidir también en este caso. La condición no es única, y depende del tipo de vendedor. Da justo para aprovechar la herencia, definir una subclase de Vendedor para cada tipo, en donde se implementa la condición que corresponde a ese tipo. P.ej. en la clase Viajante se implementa la condición definida para viajantes, y así.
Esto nos sugiere que conviene que la pregunta se le haga[3] **(1)** al vendedor. Digamos **(2)** que el mensaje sea `puedeTrabajarEn`, que lleva **(3)** un parámetro que es la ciudad. Este método **(4)** devuelve un booleano. Acá tenemos "las cuatro cosas".

Con esta definición, podemos escribir el código de `CentroDeDistribucion`

```
public boolean puedeCubrirCiudad(Ciudad ciudad) {
  return this.getVendedores().stream()
     .anyMatch(vend -> vend.puedeTrabajarEn(ciudad));
}
```

---

[1] Conviene que sea así, y no p.ej. pedirle al vendedor el conjunto de las ciudades en las que puede trabajar. Ver la sección sobre "*Pedir 'los que cumplen' o preguntar 'si alguien cumple'* ".

[2] Hay otra posibilidad, que puede servir para escenarios más complejos: preguntarle a un tercer objeto, que no sea ni la ciudad ni el vendedor. Una opción es que el tercer objeto sea una **Strategy** que modela a la condición, de forma tal que p.ej. un vendedor pueda cambiar dinámicamente el criterio sobre en qué ciudad/es puede o no trabajar.

[3] **Importante**
Podría pasar que en un caso más complejo, decidamos preguntarle a la ciudad en lugar de al vendedor, a pesar que la condición depende del vendedor. En el Centro quedaría

```
<codigo anterior>.anyMatch(vend -> ciudad.puedeTrabajar(vend));
```

¿Vale hacer esto? Sí. Alcanza con que la ciudad delegue inmediatamente en el vendedor

```
public boolean puedeTrabajar(Vendedor vend) { return vend.puedeTrabajarEn(this); }
```

En general, no es necesario que el objeto al que se le pregunta sea el mismo que tiene el código que la resuelve. Cuando estos objetos son distintos, muchas veces aparece un método en el que hay un mensaje a otro objeto con `this` como parámetro.

Del lado de los vendedores[4] queda

```
public abstract class Vendedor {
  public abstract boolean puedeTrabajarEn(Ciudad ciudad)
  // ... otros métodos
}

public class VendedorFijo extends Vendedor {
  @Override
  public boolean puedeTrabajarEn(Ciudad ciudad) {
    // ... implementación de la condición para vendedores fijos
  }
  // ... otros métodos
}

public class Viajante extends Vendedor {
  @Override
  public boolean puedeTrabajarEn(Ciudad ciudad) {
    // ... implementación de la condición para viajantes
  }
  // ... otros métodos
}

public class ComercioCorresponsal extends Vendedor {
  @Override
  public boolean puedeTrabajarEn(Ciudad ciudad) {
    // ... implementación de la condición para comercios corresponsales
  }
  // ... otros métodos
}
```

Listo, hasta acá contamos. Para cerrar la implementación, hay que ver en el enunciado cuáles son las condiciones particulares para cada tipo de vendedor.

## Conclusiones 1 - cómo llegamos a la solución

En esta sección se describe una solución a los dos puntos planteados, indicando qué clases intervienen, y qué métodos le agregamos a cada una.
Pero también (y no menos importante) dimos pautas sobre **cómo se llegó** a esta solución.
Quiero destacar dos aspectos.

1. Se empezó a pensar a partir del requerimiento. Esto es súper importante.
   Está bien darle una leida en general a un enunciado, pensando qué clases incluir, y qué métodos poner en cada una.
   Pero es el requerimiento el que te da la guía precisa, el punto de partida para elegir entre varias opciones, o para fijar las ideas que pueden quedar imprecisas en la lectura del enunciado.

[4] En un escenario más complejo, la condición podría depender de *la combinación* entre características del vendedor y de la ciudad. P.ej. podría distinguirs entre ciudades de llanura y de montaña, y que las condiciones para poder trabajar dependan del tipo de viajante **y también** del tipo de ciudad: una condición para vendedor fijo en ciudad de llanura, una distinta para vendedor fijo en ciudad de montaña, y así para viajantes y para comercios corresponsales. En un caso así, puede ser útil la técnica del *double dispatch* … de la que se encuentran fácilmente descripciones online.

Una misma situación admite varios modelos, lo que nos va orientando para decidir es para qué queremos armar un modelo. Eso te lo dan los requerimientos.

2. Se buscaron **responsabilidades**, y el armado de la solución fue guiado por las responsabilidades que fueron apareciendo.
   Una responsabilidad es algo que algún objeto tiene que responder (si es una consulta) o hacer (si es una orden).
   Cuando se detecta una responsabilidad, ayuda preguntarse "las cuatro cosas". Repasemos cuáles son para una consulta: **(1)** a qué objeto le pregunto **(2)** con qué mensaje **(3)** con qué parámetros, o sin parámetros, y **(4)** qué devuelve, pensando en un tipo.

## Conclusiones 2 - qué describimos, y qué no

Para ninguno de los dos puntos se mostró el código completo.
En el punto 3, de las certificaciones sabemos solamente que son objetos que entienden el mensaje getPuntaje(). En el punto 4, no nos metimos con las condiciones para que un vendedor pueda trabajar en una ciudad.

Esto no impide que podamos afirmar que estamos mostrando una solución a cada punto. Y creemos que el lector no tendría mucha dificultad para, a partir de lo que está descripto, llegar al código completo testeable.

Si la solución no incluye todo el código, entonces ¿**a qué estamos llamando "la solución"**, qué es lo que estamos describiendo?
Lo que sí incluimos son: las responsabilidades que encontramos, la respuesta a "las cuatro preguntas", y cómo se enganchan los métodos que definimos para solucionar cada punto.

Digo más. Al describir el método `puedeCubrirCiudad` en `CentroDeDistribucion`, lo que en rigor queremos expresar es que la condición es que algún vendedor responda `true` a `puedeTrabajarEn` con la ciudad como parámetro. Lo decimos en Java porque queda cortito y se entiende. P.ej. si nos hubíeramos olvidado del `stream()`, se entendería igual, aunque no compile.

Esto nos acerca a lo que llamamos "diseño". Se trata de identificar qué componentes va a tener mi sistema, y cómo se relacionan. En un diseño con objetos, lo que identificamos es qué clases y objetos vamos a tener, qué métodos/mensajes incluimos en cada uno, y cómo se comunican entre sí.

## Conclusiones 3 - consecuencias del diseño

Al distribuir las responsabilidades, también estamos separando el alcance de lo que hace cada objeto, o grupo de objetos.
Repasemos: en el punto 4

- **la empresa** se encarga de saber qué centros hay que considerar, y de responder la consulta inicial.
- cada **centro de distribución**, de resolver la condición sobre si cubre o no una ciudad, y
- **los vendedores**, de responder si pueden o no trabajar en una ciudad.

Tener esto en claro ayuda a entender, si hay que hacer alguna modificación, en dónde hay que tocar. En este caso, una cosa que podría cambiar es la condición para que un centro cubra una ciudad.: podría exigirse que al menos dos vendedores puedan trabajar en la ciudad, que la ciudad esté a no más de 500 km. del centro, etc.. Incluso podrían aparecer distintos tipos de centro de distribución, cada uno con su criterio. Y más etcéteras.

En la forma en que armamos la solución, en nuestro diseño, está claro que para implementar este cambio, hay que arrancar mirando la clase `CentroDeDistribucion`. En principio, ni vendedores, ni empresas, ni ciudades deberían verse afectados.
Lo mismo si aparece un error. Definimos tres responsabilidades, que son tres cuestiones que se pueden pensar, y testear, por separado. De esta forma es más fácil encontrar, aislar y corregir un error.

Todas estas son ventajas de la división de tareas. La programación con objetos, **bien usada**, proporciona entidades (los objetos, que se agrupan en clases) que se prestan bien para organizar las tareas y el código.

Para comparar, esta es la estructura de una resolución del punto 4, en Java, separando en objetos, pero sin separar las responsabilidades. Este método estaría en `Empresa`.

```
public Set<CentroDeDistribucion> getCentrosQuePuedenCubrir(Ciudad ciudad) {
  Set<CentroDeDistribucion> losCentros = new HashSet<>();
  for (CentroDeDistribucion centro : this.getCentrosDeDistribucion()) {
    boolean algunoPuede = false;
    for (Vendedor vend : centro.getVendedores()) {
      boolean condicion;
      if (vend.esVendedorFijo()) {
        // condicion para vendedores fijos
      } else if (vend.esViajante()) {
        // condicion para viajantes
      } else if (vend.esComercioCorresponsal()) {
        // condicion para comercios corresponsales
      }
      if (condicion) { algunoPuede = true; }
    }
    if (algunoPuede) {
      losCentros.add(centro);
    }
  }
  return losCentros;
}
```

Perdemos lo que dijimos de que es más fácil encarar un cambio y la corrección de un error. Además, ahora si necesitamos saber para otro requerimiento si un vendedor puede o no trabajar en una ciudad, no podemos reusar el código que nos quedó acá adentro.
A esto nos referimos con la programación con objetos **bien usada**.

# Pedir “los que cumplen” o preguntar “si alguien cumple”

**(con algo sobre métodos abstractos, atributos `static` y operador ternario )**

## Enunciado

En un ejercicio (Oktubrefest, PO1UNQ, guía 9) se pide modelar los gustos de algunas personas, respecto de las marcas de cerveza, y carpas donde se vende cerveza.

Sobre las marcas se dice lo siguiente (se quita información que no es relevante para lo que se cuenta aquí):

> Existen varias marcas de cerveza. Están las marcas de cerveza rubia (como la Corona), las marcas de cerveza negra (como la Guiness), y las marcas de cerveza roja (como la Hofbräu). De cada marca se sabe su contenido de lúpulo, o sea, cuántos gramos de lúpulo por litro llevan.
> Un dato importante relacionado con cada marca de cerveza es su *graduación*, que es el porcentaje de alcohol en volumen. P.ej. para una marca de 10 puntos de graduación, tendremos 0,1 litro de alcohol por litro de cerveza.
> Se sabe que cada marca de cerveza rubia tiene una graduación diferenciada. La graduación de una marca de cerveza negra se calcula como el mínimo entre la graduación reglamentaria y el doble de su contenido de lúpulo. La graduación reglamentaria es mundial, o sea que es única para todas las marcas de cerveza negra del mundo y puede cambiar con el tiempo.
> La cerveza roja se hace con métodos similares a la cerveza negra, pero resulta más fuerte: su graduación es un 25% superior al cálculo descripto para la cerveza negra.

Transcribimos ahora la información relevante sobre las personas:

> De cada persona interesará saber qué marcas de cerveza le gustan. Se sabe que los belgas toman sólo cerveza con más de 4 gramos de lúpulo por litro, a los checos les gustan las cervezas de más de 8 puntos de graduación, a los alemanes les gustan todas.

Finalmente, cada carpa vende una marca de cerveza. Una persona va a querer entrar a una carpa, si le gusta la marca de cerveza que vende la carpa. El manejo de los gustos cervezales de cada persona se usa para esto.

## Marcas y personas

A cualquier marca de cerveza se le tiene que poder preguntar la graduación. O sea, todas tienen que responder al **mensaje** `getGraduacion()`. Pero la forma de calcular la graduación, o sea el **método**, es distinto para cada tipo de cerveza. Acá el polimorfismo calza justo.
Como hay características comunes (el contenido de lúpulo), vamos a usar **herencia**: tendremos una superclase abstracta MarcaDeCerveza, y una clase concreta por cada "color" de cerveza. La superclase nos queda así.

```
public abstract class MarcaDeCerveza {
    private double contenidoDeLupulo;

    public double getContenidoDeLupulo() { return this.contenidoDeLupulo; }
    public void setContenidoDeLupulo(double cont) {
        this.contenidoDeLupulo = cont;
    }
    public abstract double getGraduacion();
}
```

Genial. Ahora pensemos en las personas. El enunciado dice "interesará saber qué marcas de cerveza le gustan". La primera intención es definir métodos de esta forma

```
    public Collection<MarcaDeCerveza> getMarcasQueLeGustan() {
        // ...codigo...
    }
```

en las clases que representan a belgas, checos y alemanes.

Pensemos en cómo lo definimos para los alemanes, que le gustan todas las cervezas. Deberíamos devolver la colección de todas las marcas de cerveza. Pero … ¿de dónde sacamos esa colección? En el enunciado no hay nada que indique que hay un objeto, ponele un catálogo (o una base de datos) a la que se le pueda pedir esa colección.

Entonces ¿no sale el ejercicio, no hay forma de hacerlo?

Stop.

El ejercicio no sale **porque lo estamos encarando mal**.

En ningún lugar se pide poder preguntarle a una persona **la colección de marcas** que le gustan. Alcanza con poder preguntarle, **dada una marca**, si le gusta o no. O sea

```
    public boolean leGusta(MarcaDeCerveza marca) {
        // ...codigo...
    }
```

Esto alcanza para poder determinar si quiere entrar a una carpa o no

```
    public boolean quiereEntrar(Carpa carpa) {
        return this.leGusta(carpa.marcaQueVende());
    }
```

Así sí sale. Para los alemanes es trivial

```
public boolean leGusta(MarcaDeCerveza marca) {
    return true;
}
```

Para los belgas tenemos

```
public boolean leGusta(MarcaDeCerveza marca) {
    return marca.getContenidoDeLupulo() > 4;
}
```

Finalmente, para los checos es así

```
public boolean leGusta(MarcaDeCerveza marca) {
    return marca.getGraduacion() > 8;
}
```

Listo, problema resuelto.

## Conclusiones hasta acá

De este código podemos sacar algunas conclusiones.

La primera es la que sugiere el título de la sección.
A veces no se puede devolver la colección de objetos que cumplen una condición, porque no se puede construir. Pero sí se puede armar un método booleano que, **dado un objeto**, indica si la condición se cumple o no. Es posible que con esto alcance, porque siempre que se pregunte, ya se sabe para qué objeto preguntar.
En este caso,

- no se puede armar la colección de marcas que le gustan a una persona, pero
- sí se puede decidir, dada una marca de cerveza, si a una persona le gusta o no, y
- con lo segundo alcanza para lo que pide el enunciado, que es poder decidir si una persona quiere entrar a una carpa o no.

Dicho de otra forma: a veces hay una alternativa entre un método que devuelve una colección de X, y un método booleano que recibe un X como parámetro (en el ejemplo X = marca de cerveza).
En el ejemplo, las alternativas son

```
public Collection<MarcaDeCerveza> getMarcasQueLeGustan() {
    // ...codigo...
}
```

```
public boolean leGusta(MarcaDeCerveza marca) {
    // ...codigo...
}
```

En estos casos, puede ser interesante pensar si las dos se pueden, y qué opción conviene.

Otra conclusión interesante es que pudimos armar el código de las clases que representan personas, sin necesidad de pensar en p.ej. la cuenta para calcular la graduación de una cerveza. Cuando estoy implementando las personas, lo único que me tiene que interesar es que a cualquier cerveza le puedo pedir la graduación, no cómo se calcula. O sea, pienso en mensajes, no en métodos.

## Marcas de cerveza

Analicemos un poco de la implementación de las marcas de cerveza.

```
public abstract class MarcaDeCerveza {
    private double contenidoDeLupulo;

    public double getContenidoDeLupulo() { return this.contenidoDeLupulo; }
    public void setContenidoDeLupulo(double cont) {
        this.contenidoDeLupulo = cont;
    }
    public abstract double getGraduacion();
}

public class MarcaDeCervezaRubia extends MarcaDeCerveza {
    private double graduacion;

    @Override
    public double getGraduacion() { return this.graduacion; }
    public void setGraduacion(double grad) { this.graduacion = grad; }
}

public class MarcaDeCervezaNegra extends MarcaDeCerveza {
    private static double graduacionReglamentaria;

    public static double getGraduacionReglamentaria() {
        return graduacionReglamentaria;
    }
    public static void setGraduacionReglamentaria(double grad) {
        graduacionReglamentaria = grad;
    }

    @Override
    public double getGraduacion() {
        double dobleDelLupulo = this.getContenidoDeLupulo() * 2;
        double porReglamento = MarcaDeCervezaNegra.getGraduacionReglamentaria();
        return Math.min(regl, dobleLupulo);
    }
}

public class MarcaDeCervezaRoja extends MarcaDeCervezaNegra {
    @Override
    public double getGraduacion() {
        return super.getGraduacion() * 1.25;
    }
}
```

Nos sirve para repasar por qué hace falta el método abstracto `getGraduacion()` en `MarcaDeCerveza`. Si no lo ponemos, el método que escribimos para los checos

```
    public boolean leGusta(MarcaDeCerveza marca) {
        return marca.getGraduacion() > 8;
    }
```

no compila. A Java no le interesa que todas las implementaciones de `MarcaDeCerveza` entiendan el mensaje, necesita que el método esté en `MarcaDeCerveza`. Si no hay ninguna implementación por defecto, pues hay que definirlo abstracto.

En esta implementación también aparecen tres elementos que sirven en otros casos, los mencionamos:

1. Se usa un atributo `static` para un valor que puede cambiar (o sea, que no se puede poner el valor en el código) pero que es el mismo para todas las instancias de una clase. En este caso, el valor de graduación reglamentaria es el mismo para todas las instancias de `MarcaDeCervezaNegra`, por eso se maneja con un atributo static (con su getter y su setter).

2. El método `getGraduacion()` de `MarcaDeCervezaNegra` incluye un *operador ternario*, en el return:

```
return (dobleDelLupulo < porReglamento)
                ? dobleDelLupulo : porReglamento;
```

esto es lo mismo que

```
if (dobleDelLupulo < porReglamento) {
    return dobleDelLupulo;
} else {
    return porReglamento;
}
```

pero un poco más corto.

3. La clase concreta `MarcaDeCervezaRoja` es subclase de `MarcaDeCervezaNegra`, que también es concreta. Eso vale, no es cierto que solamente se puede heredar de clases abstractas. La condición es que “sea parecido pero distinto”.
Las marcas rojas son parecidas a las negras, con una sola diferencia. Va herencia, listo, sin importar si la superclase es abstracta o concreta.