Oktoberfest

Ein Prosit! En Münich-Alemania, todos los años se festeja la Oktoberfest. Para esta gran feria de comidas, entretenimientos y mucha cerveza nos piden a construir un programa en objetos que modele el comportamiento de las personas en la fiesta.

Al entrar en la Oktoberfest se pueden encontrar enormes carpas cerveceras, en donde muchísima gente se reune a… bueno... tomar cerveza. Queremos controlar la entrada a estas carpas dependiendo de la disponibilidad de la carpa y los gustos del público.

Las carpas cerveceras tienen un límite de gente admitida, algunas tienen una banda para tocar música tradicional, y por supuesto que todas venden jarras de cerveza. De cada jarra de cerveza sabemos su capacidad en litros y de qué marca es la cerveza. De cada marca de cerveza se sabe su graduación alcohólica. Cada carpa sirve jarras de cerveza de sólo una marca y siempre de la misma capacidad (que depende de cada carpa).

De cada persona se sabe su peso, la jarras de cerveza que compró hasta el momento, si le gusta escuchar música tradicional o no y su aguante. Una persona está ebria si la cantidad de alcohol en sangre que tiene multiplicado por su peso supera su aguante.

Además, de cada persona interesará saber qué marcas de cerveza le gustan. Se sabe que los belgas toman sólo cerveza de su país, a los checos les gustan las cervezas de más de 8% de alcohol, a los alemanes… les gustan todas.

Se requiere:

1.

a. Saber cuántos litros de alcohol aporta una jarra de cerveza. Ej: una jarra de cerveza de medio litro de la marca ‘Hofbräu’ (que tiene 8% de graduación alcohólica) aporta 0,5 * 0,08 = 0,04 litros de alcohol.

b. Saber el total de litros de alcohol que ingirió una persona (en base a las jarras de cerveza que compró).

c. Saber si una persona está ebria.

2.

a. Saber si una persona quiere entrar a una carpa, es decir, si la carpa vende una marca de cerveza que a él le guste y si cumple su preferencia sobre que haya o no haya música.

b. Saber si una carpa deja ingresar a una persona, o sea, si dejándola entrar no supera su límite de personas y si la persona no está ebria.

c. Saber si una persona puede entrar a una carpa, es decir, si quiere entrar a la carpa y la carpa lo deja entrar.

3. Hacer que una persona entre en una carpa, si puede. Cuando la persona entra en la carpa, se compra una jarra de cerveza y, por supuesto, la carpa queda con menos capacidad. Si no puede entrar en la carpa, se debe tirar error.

Nota: las carpas no tienen un “stock” de jarras de cerveza ya listas para usar, se debe crear una nueva en base al tamaño y marca que vende la carpa.

4. Saber cuantos ebrios empedernidos hay dentro de una carpa. Los ebrios empedernidos son los ebrios que sólo compraron jarras de 1 litro.

5. Dentro de la feria también hay juegos con premios que dependen de la habilidad -o más bien, sobriedad- del jugador (por ejemplo: tiro al blanco, medir la fuerza golpeando con un martillo, dardos, etc). De los juegos sabemos el límite de alcohol en sangre sugerido para que tenga sentido que una persona se anime a jugarlo. Se desea saber si una persona puede

entrar a jugar a un juego. Una persona puede participar de un juego si ya consumió alguna cerveza (¡para el coraje!) pero todavía no llegó al límite de alcohol en sangre sugerido por el juego.

6. Algunas carpas sólo permiten el ingreso si la persona, además de cumplir los requisitos ya descriptos, también hizo una reserva con anticipación. Además, las carpas con reserva reciben a su público con una jarra de 0,3 litros gratis (aparte de la que el que alemán que ingresa vaya a comprar). Incorporar este nuevo tipo de carpa.

7. Dada una colección de atracciones de la feria (juegos y carpas), saber a cuáles no puede entrar una persona.

Una Posible Resolución

Siempre es una buena práctica empezar por el workspace, porque así nos concentramos en los requerimientos y nos ayuda a pensar a que objeto le quiero mandar que mensaje (o sea, quien es el responsable de resolver el problema).

Vamos a ir enfocándonos punto por punto

1)
*a. Saber cuántos litros de alcohol aporta una jarra de cerveza. Ej.: una jarra de cerveza de medio litro de la marca 'Hofbräu' (que tiene 8% de graduación alcohólica) aporta 0,5 * 0,08 = 0,04 litros de alcohol.*

Si pensamos en el workspace y en la sintaxis de smalltalk la pregunta que surge es ¿a qué objeto le mando qué mensaje? Fijense que en este momento no me importa ni siquiera como es el cálculo que hay que hacer, desde el workspace mi problema es otro.
Y la verdad que no sé a quién mandárselo pero si releemos el enunciado en un momento menciona jarras de cerveza y dice: *De cada jarra de cerveza sabemos su capacidad en litros y de qué marca es la cerveza.* Entonces el objeto jarra de cerveza que tiene una capacidad en litros parece ser un buen lugar donde tener el mensaje.
Imagino una jarra de cerveza y pienso en como me gustaría usarlo:

```
jarraDeStella alcoholAportadoEnLitros.
```

Antes de seguir, fíjense que me tomo el tiempo de escribir nombres descriptivos (por mas largos que sean). El selector de un mensaje debe darme suficiente información para saber que es lo que hace, sin necesidad de saber como esta implementado; es decir, debe ser **expresivo**, que es algo sobre lo que insistimos mucho en la materia[1]

Volviendo al requerimiento:

Decidí mandarle el mensaje a la jarra, además porque ella tiene la *información mínima necesaria* para resolver el problema: *De cada jarra de cerveza sabemos su capacidad en litros y de qué marca es la cerveza.* Entonces, `jarraDeStella` seria una instancia de la clase `Jarra`, para que entienda el mensaje voy a codificar el método en la clase Jarra. Se me ocurrieron dos alternativas asi que vamos a codificar ambas:

Variante 1: Accediendo directamente a las variables de instancia

```
#Jarra
>>alcoholAportadoEnLitros
^capacidadEnLitros* marca graduacionAlcoholica
```

Analicemos el código, ¿Cuántos mensajes estamos enviando y a que objetos?
2 mensajes:

[1] Recuerden: su código debe ser apto para seres humanos ^^

- * al objeto referenciado por capacidadEnLitros
- graduacionAlcoholica al objeto refenciado por marca

Recuerden que “Solo puedo usar directamente las variables propias, NUNCA vamos a acceder a los atributos de otros objetos porque la manera de comunicarse con los objetos es mandándole mensajes (el hecho de que cada objeto no exponga sus atributos al resto del programa es lo que llamamos **encapsulamiento**)”

Variante 2: usando accessors[2]

```
#Jarra
>>cuantosLitrosDeAlcoholAportas
^self capacidadEnLitros * self marca graduacionAlcoholica
```

Hagamos el mismo análisis, ¿Cuántos mensajes estamos enviando y a qué objetos?
4 mensajes:

- * al objeto resultado de la evaluación de la expresión `self capacidadEnLitros`
- graduacionAlcoholica al objeto resultado de la evaluación de la expresión `self marca`
- `capacidadEnLitros` al objeto que recibió el mensaje (`self`)
- `marca` al objeto que recibió el mensaje (`self`)

La diferencia es con el caso anterior es que no estoy accediendo a las variables directamente, estoy usando **mensajes**, esto me da la libertad de que si el día de mañana cambia la forma de calcular la capacidad en litros de una jarra y fui consistente (siempre use el accesor) solamente cambiamos en un solo lugar. Para que este código funcione el objeto tiene que entender los mensajes, por lo que deben estar definidos los métodos. Por simplicidad en los parciales (y a veces en el pizarrón), no codificamos los accessors, pero en la máquina si hay que hacerlo.

b. Saber el total de litros de alcohol que ingirió una persona (en base a las jarras de cerveza que compró).

```
williamWallace totalEnLitrosDeAlcoholIngeridos.
```

Esto depende de la cantidad de jarras que haya tomado la persona; ahora bien, primera decisión importante que tomar: ¿de que manera debe conocer una persona las jarras que tomo? ¿debe conocer a las jarras, o solo contarlas?
Cada jarra sabe cuantos litros de alcohol aporta, lo cual me lleva a pensar que no me alcanza con contarlas nomas (es decir, no puedo usar una variable como acumulador); entonces, como quiero mandarles mensajes a las jarras como conjunto, voy a decir que las personas conocen a una colección de jarras.

Variante 1: Usando #collect: y #sum

```
#Persona
>>cantidadEnLitrosDeAlcoholIngeridos
```

[2] O “accesors” o *getters y setters*. El nombre *Accesors* es una convención que se usa entre los smalltalkeros para denominar a los mensajes que settean (asignan) una variable -*setters o mutators*- y devuelven el objeto referenciado por una variable -*getters*-.

```
^(self jarrasTomadas collect: [:jarra | jarra
cuantosLitrosDeAlcoholAportas]) sum.
```

El mensaje #sum es propio de los dialectos Squeak / Pharo. En otros, podría no existir (Dolphin, Visual Age, etc.) y deberíamos usar otra variante (o implementar nosotros #sum, lo que más nos guste). También existe el mensaje #sum: que recibe un bloque y hace esto mismo que acabamos de hacer con #collect: y #sum.

Variante 2: Usando #sum:

```
#Persona
>> cantidadEnLitrosDeAlcoholIngeridos
^self jarrasTomadas sum: [:jarra | jarra
cuantosLitrosDeAlcoholAportas].
```

Variante 3: Usando #inject: into:

```
#Persona
>> cantidadEnLitrosDeAlcoholIngeridos
^self jarrasTomadas inject: 0 into: [:acum :jarra | acum + jarra
cuantosLitrosDeAlcoholAportas].
```

c. Saber si una persona está ebria.

```
williamWallace estasEbrio.
```

De cada persona se sabe su peso, la jarras de cerveza que compró hasta el momento, si le gusta escuchar música tradicional o no y su aguante. Una persona está ebria si la cantidad de alcohol en sangre que tiene multiplicado por su peso supera su aguante.

Entonces, de nuevo, la responsabilidad de decirme si una persona esta ebria o no es de la persona:

```
#Persona
>>estasEbrio
^(self cantidadEnLitrosDeAlcoholIngeridos * self peso ) > self
aguante.
```

Siendo #peso y #aguante los getters de las variables de instancia peso y aguante, respectivamente.

2.
a. Saber si una persona quiere entrar a una carpa, es decir, si la carpa vende una marca de cerveza que a él le guste y si cumple su preferencia sobre que haya o no haya música.

williamWallace queresEntrarA: **carpaHeineken**

```
#Persona
```

```
>>queresEntrarA: unaCarpa
^(self teGusta: unaCarpa cervezaVendida) and: [unaCarpa
cumplisPreferenciaDe: self]
```

También podría haber usado el mensaje #&

```
>>queresEntrarA: unaCarpa
^(self teGusta: unaCarpa cervezaVendida) & ( unaCarpa
cumplisPreferenciaDe: self)[3]

#Carpa
>>cumplisPreferenciaDe: unaPersona
^unaPersona leGustaLaMusicaTradicional and: [self
tenesBandaTradicional].
```

Se sabe que los belgas toman sólo cerveza de su país, a los checos les gustan las cervezas de más de 8% de alcohol, a los alemanes… les gustan todas.

Bien, resulta que tenemos comportamientos distintos según de que nacionalidad sea la persona... Pero, son todas personas! Y tienen cosas en común! Y además, las quiero tratar a todas indistintamente... Podría preguntarle a cada persona de que nacionalidad es, algo así como:

```
#Persona
>>teGusta: unaCerveza
self nacionalidad = 'Belga' ifTrue:[ ^unaCerveza paisDeOrigen ==
'Belgica'].
self nacionalidad = 'Checo' ifTrue: [^unaCerveza graduacionAlcoholica
>8].
self nacionalidad = 'Aleman' ifTrue: [^true].
```

Pero en realidad, eso es una manera muy fea de encarar el problema, porque **estoy repitiendo código**, y porque en realidad, cada persona debe saber decirme cuando le gusta una cerveza, según su nacionalidad...
Entonces, recapitulemos: tengo comportamiento en común; una **abstracción**, una generalización; que es la Persona, y además tengo comportamiento particular de cada tipo de persona... entonces, podemos usar **herencia[4], delegando** en las subclases la responsabilidad de responder si a la persona le gusta una cerveza, con lo cual las instancias de cada subclase serían **polimórficas** para el mensaje #queresEntrarA:.

---

[3] La diferencia entre #& y #and: es que como #and: recibe un bloque, solo se evalúa la segunda condición si la primera es verdadera:

```
#True >> and: aBlock                    #False>>and:aBlock
^aBlock value.                          ^false
>>& aBoolean                           >>& aBoolean
 ^aBoolean                              ^false
```

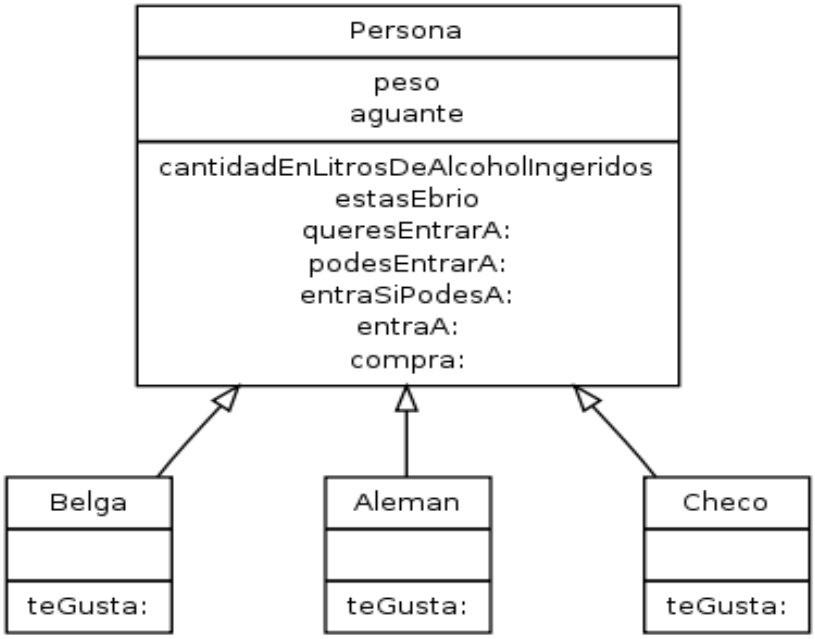
[4]Recordemos que la herencia es una forma de compartir código entre clases. Usar herencia se trata de encontrar una generalización, una abstracción rica que permita modelar un concepto más general en donde pueda ubicar el comportamiento común que encuentro entre varias clases.

:

```
#Belga
>>teGusta: unaCerveza
^unaCerveza paisDeOrigen = 'Belgica'

#Checo
>>teGusta: unaCerveza
^unaCerveza graduacionAlcoholica > 8

#Aleman
>>teGusta: unaCerveza
^true.
```



Y así cada uno sabe cuando le gusta una cerveza... Esta solución es mas extensible: si el día de mañana aparecen los argentinos, que solo toman cervezas de calidad premium (?) solo tengo que agregar una subclase de persona y definirle el método `#teGusta:` y las personas siguen siendo polimórficas.

Nótese que no tiene sentido tener instancias de la clase Persona ahora, porque una instancia de Persona no entendería el mensaje #teGusta:, necesario para poder resolver #queresEntrarA:.
A las clases de las que no tiene sentido tener instancias las llamamos “clases abstractas”.

b. Saber si una carpa deja ingresar a una persona, o sea, si dejándola entrar no supera su límite de personas y si la persona no está ebria.

carpaHeineken dejasEntrarA: **williamWallace**

```
#Carpa
>>dejasEntrarA: unaPersona
^self noSuperaElLimiteSiEntraUnoMas and: [unaPersona estasEbrio not].

>>noSuperaElLimiteSiEntraUnoMas
^self cantidadDePersonasAdentro < self limite.
```

Por ahora, la cantidad de “personas adentro” podría ser simplemente un número, un “contador de personas”, que vamos a tener en el atributo “cantidadDePersonasAdentro”.

```
>>cantidadDePersonasAdentro
^cantidadDePersonasAdentro
```

c. Saber si una persona puede entrar a una carpa, es decir, si quiere entrar a la carpa y la carpa lo deja entrar.

williamWallace podesEntrarA: **carpaHeineken**

```
#Persona
>>podesEntrarA: unaCarpa
^self queresEntrarA: unaCarpa and: [unaCarpa dejasEntrarA: self]
```

3. Hacer que una persona entre en una carpa, si puede. Cuando la persona entra en la carpa, se compra una jarra de cerveza y, por supuesto, la carpa queda con menos capacidad. Si no puede entrar en la carpa, se debe tirar error.
Nota: las carpas no tienen un “stock” de jarras de cerveza ya listas para usar, se debe crear una nueva en base al tamaño y marca que vende la carpa.

williamWallace entraSiPodesA: **carpaHeineken**

```
#Persona
>>entraSiPodesA: unaCarpa
^(self podesEntrarA: unaCarpa)
      ifTrue: [self entraA: unaCarpa]
      ifFalse:[self error: 'La persona no puede entrar en la carpa'].
```

Quiero detenerme un segundo en el mensaje `#error:` ; este mensaje lo entienden todos los objetos. Alguno me va a decir: che, ¿no es más fácil hacerme mi propio mensaje de error `#avisameElError` que devuelva un string? El chiste de usar `#error:` es que este mensaje no solo nos muestra un cartelito, sino que además interrumpe el flujo de ejecución, es decir, interrumpe el envío de mensajes. Entonces, si yo uso un mensaje propio que solo devuelve un string, si algo salio mal, yo no me entero... Y si en algún momento de mi programa hay algo que depende de ese envío de mensajes que salió mal, seguramente no se comporte como estoy esperando y me va a costar muuuuuuuucho saber por qué.

Teniendo en cuenta que el mensaje #error: corta el flujo de ejecución, también se podría haber escrito de esta manera:

```
#Persona
>>entraSiPodesA: unaCarpa
^(self podesEntrarA: unaCarpa)
      ifFalse: [self error: 'La persona no puede entrar en la carpa'].
     self entraA: unaCarpa
```

Si la persona no puede entrar a la carpa, el error va a cortar la ejecución del método. Por lo tanto, si la última línea se ejecuta es que no hubo ningún error.

Dicho eso, podemos seguir:

```
>>entraA: unaCarpa
self compra: unaCarpa nuevaJarraDeCervezaPropia.
unaCarpa reduciCapacidad.
```

Otra opción puede ser decirle a la carpa que una persona entró, y que la carpa haga lo que corresponda (lo cual es una decisión muy saludable)

```
>>entraA: unaCarpa
self compra: unaCarpa nuevaJarraDeCervezaPropia.
unaCarpa entro:self.

>>compra: unaJarraDeCerveza
self agregaJarra: unaJarraDeCerveza.

>>agregaJarra: unaJarra
self jarrasTomadas add: unaJarra

#Carpa
>>entro: unaPersona
self reduciCapacidad.

>>reduciCapacidad
cantidadDePersonasAdentro:= cantidadDePersonasAdentro + 1
```

Y ahora, me voy a #Carpa

Variante 1: usando metodos de clase

```
#Carpa
>>nuevaJarraDeCervezaPropia
^Jarra nuevaJarraDeMarca: self marcaVendida yCapacidad: self
capacidadJarra.

#Jarra
>>nuevaJarraDeMarca: unaMarca yCapacidad: unaCapacidad (MC)
```

```
|jarra|
jarra:= self new.
jarra capacidad: unaCapacidad.
jarra marca: unaMarca.
^jarra
```

Variante 2: mandandole new directamente la clase

```
#Carpa
>>nuevaJarraDeCervezaPropia
^Jarra new capacidad: self capacidadJarra; marca: self marcaVendida.
```

Nótese el uno de “punto y coma”: el punto y coma se usa para mandarle otro mensaje más al mismo receptor del primer mensaje. En este caso, primero se le manda el mensaje #capacidad: a la nueva jarra, y luego el mensaje #marca: a ese mismo objeto.

Variante 3: delegando la creación de una nueva jarra a la marca

```
#Carpa
>>nuevaJarraDeCervezaPropia
^self marcaVendida nuevaJarraDeCapacidad: self capacidadJarra

#Marca
>>nuevaJarraDeCapacidad: unaCapacidad
^Jarra new capacidad: unaCapacidad; marca: self.
```

4. Saber cuantos ebrios empedernidos hay dentro de una carpa. Los ebrios empedernidos son los ebrios que sólo compraron jarras de 1 litro.

carpaHeineken cuantosEbriosEmpedernidosTenes.

Bien, vuelvo a tomar la decisión de que la carpa conozca a la gente que tiene adentro mediante una colección, ya que necesito mandarle mensajes a las personas, no solo contarlas. Una carpa va a conocer a las personas que tiene adentro mediante el atributo “personasAdentro”. Entonces:

Un error común suele ser escribir directamente:

```
#Carpa
>>cuantosEbriosEmpedernidosTenes
^self personasAdentro
     select: [:persona | persona jarrasTomadas allSatisfy: [: jarra |
          jarra capacidadEnLitros>1] ]
```

Si bien eso funciona, y en definitiva es lo que queremos hacer, no esta bueno que la lógica que determina cuando una persona es ebria empedernida este dentro del bloque del select:, porque eso es responsabilidad de la persona saberlo, no de la carpa... En esta solución, fíjense que el objeto carpa esta sabiendo muchas cosas de las personas: que tienen una colección de jarras (porque les manda el mensaje allSatisfy:), que las jarras conocen su capacidad... Y

además, si el día de mañana el requerimiento cambia y por ejemplo las personas no tienen mas una colección de jarras, hay que cambiar todo.... Un bajón :(Por eso, esta bueno **delegar** la responsabilidad de saber todo eso en el objeto que le corresponde, en este caso la persona.

```
#Carpa
>>cuantosEbriosEmpedernidosTenes
^self personasAdentro select: [:persona | persona sosEbrioEmpedernido]

#Persona
>>sosEbrioEmpedernido
^self jarrasTomadas allSatisfy: [:jarra | jarra capacidadEnLitros >1].
```

Ahora... Yo recuerdo el segundo punto, cuando le preguntaba a una carpa si una persona podía entrar, que tenía este método:

```
#Carpa
>>noSuperaElLimiteSiEntraUnoMas
^self cantidadDePersonasAdentro < self limite.

#Carpa
>>cantidadDePersonasAdentro
^cantidadDePersonasAdentro
```

Peeeeeeeeeeeeeeeeeeeero, en ese momento yo solo necesitaba contar a las personas... Ahora que el requerimiento cambió, tengo dos opciones: o dejo el "contador" y la colección, o dejo solo la colección. Mi voto es **dejar sólo la colección**, porque las colecciones saben decirme cuántos elementos tienen, y porque si dejo el contador tengo que asegurarme de mantener la consistencia, y además sería tener información redundante.
Entonces, con el nuevo diseño, ese método quedaria así:

```
#Carpa
>>cantidadDePersonasAdentro
^ personasAdentro size
```

Y además, el tema de reducir la capacidad, ahora que tengo una colección, no necesito tener una variable capacidad e ir actualizándola.

```
#Carpa
>>entro: unaPersona
self reducirCapacidad.
self personasAdentro add: unaPersona.
```

El método #reducirCapacidad ya no tiene sentido y lo "podemos borrar".

5. Dentro de la feria también hay juegos con premios que dependen de la habilidad -o más bien, sobriedad- del jugador (por ejemplo: tiro al blanco, medir la fuerza golpeando con un martillo, dardos, etc). De los juegos sabemos el límite de alcohol en sangre sugerido para que tenga sentido que una persona se anime a jugarlo. Se desea saber si una persona puede entrar a jugar a un juego. Una persona puede participar de un juego si ya consumió alguna cerveza (¡para el coraje!) pero todavía no llegó al límite de alcohol en sangre sugerido por el

juego.

williamWallace podesJugar: **tiroAlBlanco**

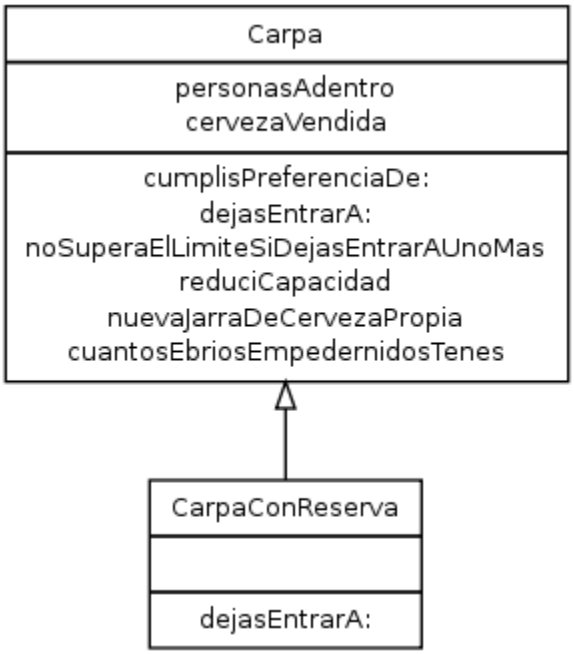
```
#Persona
>>podesJugar: unJuego
^self consumisteAlgunaCerveza and: [ self
cantidadEnLitrosDeAlcoholIngeridos < unJuego
limiteDeAlcoholEnSangreSugerido].

>>consumisteAlgunaCerveza
^self jarrasTomadas notEmpty. "Porque si consumio alguna, la colección
no esta vacia"

#Juego
>>limiteDeAlcoholEnSangreSugerido
^limiteDeAlcoholEnSangreSugerido
```

6. Algunas carpas sólo permiten el ingreso si la persona, además de cumplir los requisitos ya descriptos, también hizo una reserva con anticipación. Además, las carpas con reserva reciben a su público con una jarra de 0,3 litros gratis (aparte de la que el que alemán que ingresa vaya a comprar). Incorporar este nuevo tipo de carpa.

Bueno, estas nuevas `carpasConReserva` hacen cosas distintas a las carpas que veníamos modelando hasta ahora... Pero también hacen cosas igual :) Eso me lleva a pensar que puedo usar herencia de nuevo: `#CarpaConReserva` es subclase de `#Carpa`. Por lo tanto, sus instancias entienden el mensaje `#dejasEntrarA:` . Pero, las carpasConReserva tienen las mismas condiciones para dejar a alguien, y **además agregan otro comportamiento.**
La herramienta para resolver este problema es hacer una **redefinición** del método en la subclase y el uso de la pseudo variable **super** (que al igual que self, referencia al objeto receptor del mensaje, pero altera el method lookup):




```
#CarpaConReserva
>>dejasEntrarA: unaPersona
^(super dejasEntrarA: unaPersona) and: [unaPersona
reservoConAnticipacion: self].

>>entro: unaPersona
super[5] entro: unaPersona
self regalaCervezaA: unaPersona.

>>regalaCervezaA: unaPersona
unaPersona agregaJarra: self nuevaJarraChica.

>>nuevaJarraChica
^self nuevaJarraDeCerveza capacidadEnLitros: 0.3.
```

#Persona
>>reservoConAnticipacion: unaCarpa
^self carpasReservadas includes: unaCarpa

7. Dada una colección de atracciones de la feria (juegos y carpas), saber a cuáles no puede entrar una persona.

Suponiendo que la colección de atracciones ya existe:

---

[5]Reminder: Solo usen *super* cuando no puedan usar *self* (porque les generaría un loop infinito).

```
#Feria
>>atraccionesProhibidasPara: unaPersona
^self atracciones reject: [:atraccion | unaPersona podesParticiparEn:
atraccion].
```

Pero acá tenemos un problema: las atracciones pueden ser juegos o carpas, y las personas les mandan mensajes distintos... entonces, para aprovechar el **polimorfismo** y tratar a las atracciones indistintamente, en lugar de delegar en la persona saber si puede entrar, lo voy a delegar en la atracción:

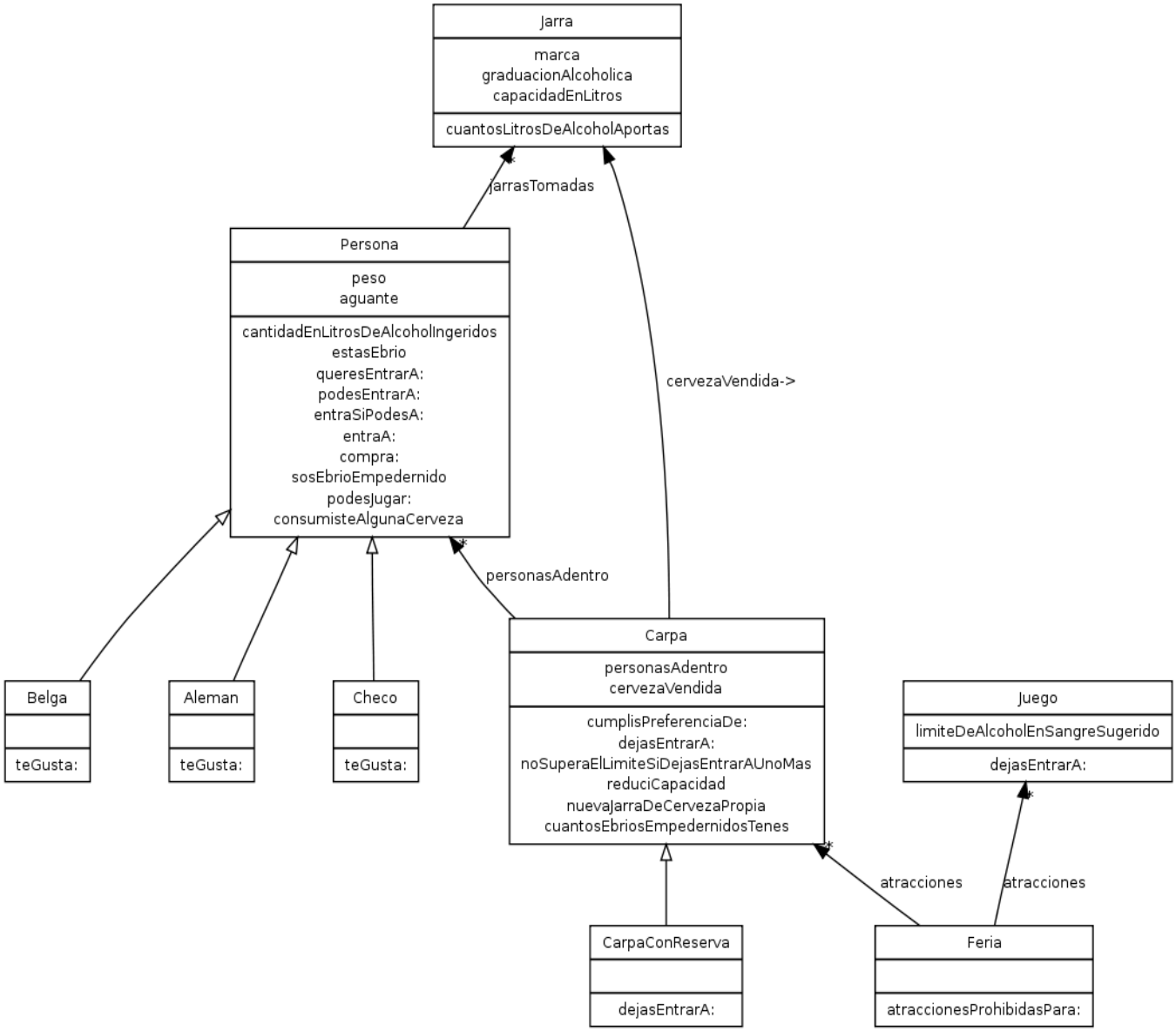
```
#Feria
>>atraccionesProhibidasPara: unaPersona
^self atracciones reject: [: atraccion | atraccion dejasEntrarA:
unaPersona].

#Juego
>>dejasEntrarA: unaPersona
^unaPersona podesJugar: self.
```

Ok, termino delegando en la persona, pero trato a las atracciones polimorficamente :)

Y para las carpas ya lo habiamos definido en puntos anteriores:

```
#Carpa
>>dejasEntrarA: unaPersona
^self noSuperaElLimiteSiEntraUnoMas and: [unaPersona estasEbrio not].
```


Jarra
marca
graduacionAlcoholica
capacidadEnLitros
cuantosLitrosDeAlcoholAportas
jarrasTomadas
Persona
peso
aguante
cantidadEnLitrosDeAlcoholIngeridos
estasEbrio
queresEntrarA:
podesEntrarA:
entraSiPodesA:
entraA:
compra:
sosEbrioEmpedernido
podesJugar:
consumisteAlgunaCerveza
cervezaVendida->
personasAdentro
Belga
teGusta:
Aleman
teGusta:
Checo
teGusta:
Carpa
personasAdentro
cervezaVendida
cumplisPreferenciaDe:
dejasEntrarA:
noSuperaElLimiteSiDejasEntrarAUnoMas
reduciCapacidad
nuevaJarraDeCervezaPropia
cuantosEbriosEmpedernidosTenes
Juego
limiteDeAlcoholEnSangreSugerido
dejasEntrarA:
atracciones
atracciones
CarpaConReserva
dejasEntrarA:
Feria
atraccionesProhibidasPara: