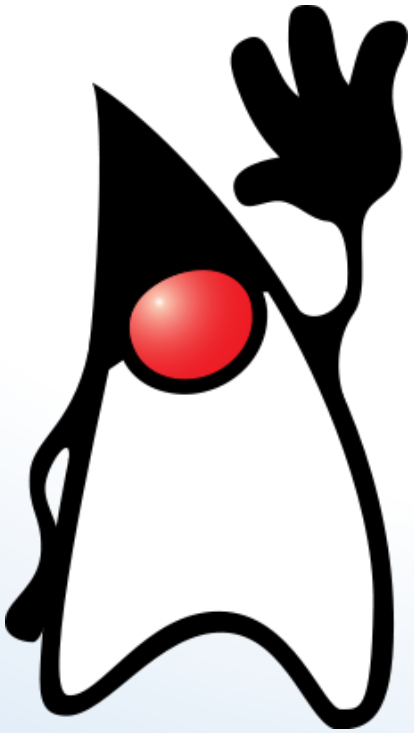
75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

.UBAfiuba
FACULTAD DE INGENIERÍA

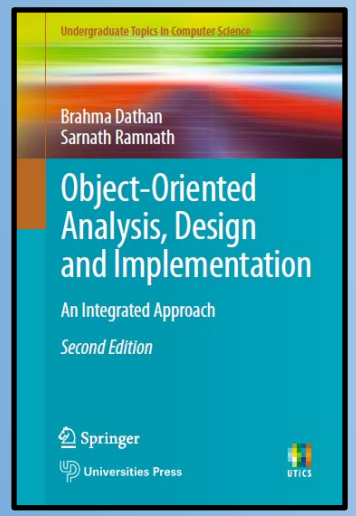
Programación Orientada a Objetos

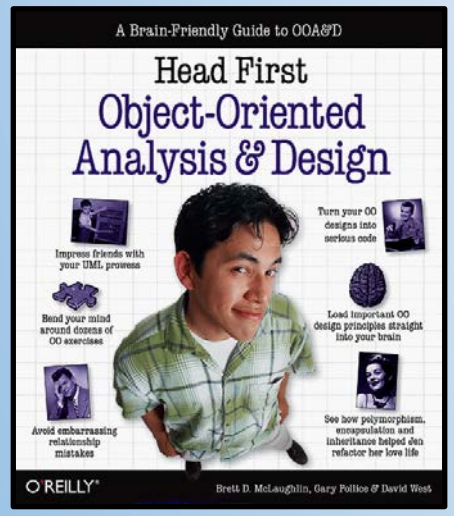


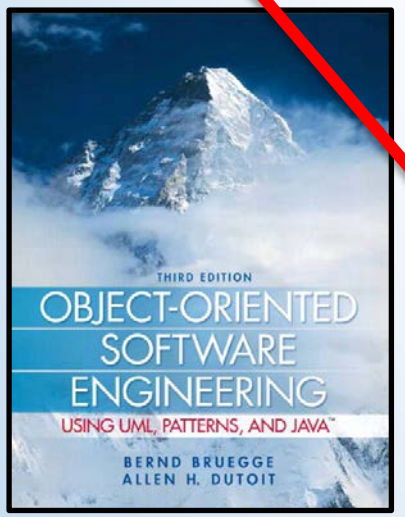
Cátedra: Corsi - Essaya - Maraggi

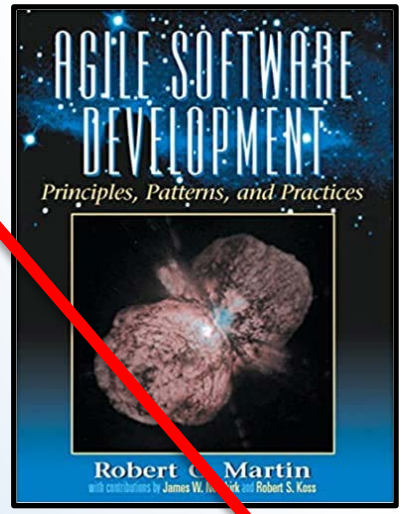
75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

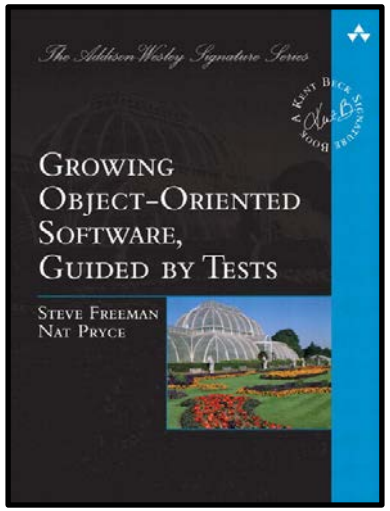
.UBAfiuba
FACULTAD DE INGENIERÍA

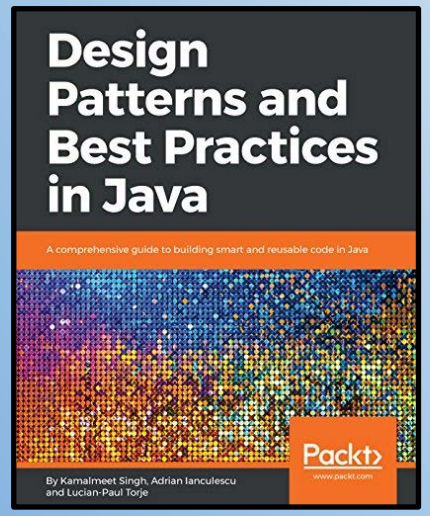


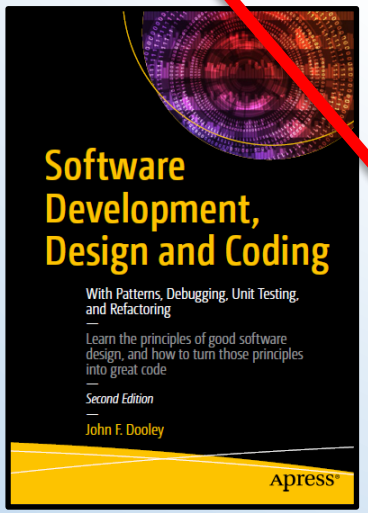


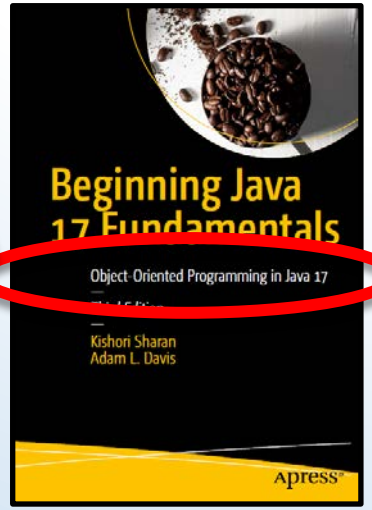












Paradigmas de Programación

- **Imperativos** (énfasis en la **ejecución de instrucciones**)
  - Programación Procedimental (p. ej. *Pascal*)
  - **Programación Orientada a Objetos** (p. ej. *Smalltalk*)
- **Declarativos** (énfasis en la **evaluación de expresiones**)
  - Programación Funcional (p. ej. *Haskell*)
  - Programación Lógica (p. ej. *Prolog*)

Los lenguajes más utilizados (por ejemplo, ***Java***) son multiparadigma: Cabe a los programadores usar el estilo de programación más adecuado para cada trabajo.

"En vez de un procesador de bits consumiendo estructuras de datos, tenemos un universo de ***objetos*** *bien comportados, cada uno de ellos pidiéndole a otro que cortésmente le realice sus variados deseos".*

Fuente: Ingalls, Daniel H. H. "Design Principles Behind Smalltalk". En: *BYTE Magazine* (Agosto, 1981). McGraw-Hill, Nueva York, p. 290

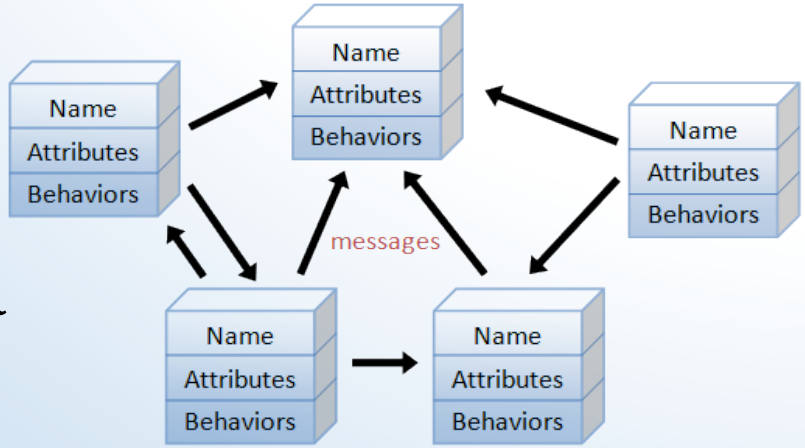
Sistemas Orientados a Objetos

- En la programación tradicional, los datos son pasivos y se les aplican procesos.
- En cambio, cuando se trabaja con objetos, son estos los que **actúan** para resolver un problema, respondiendo a estímulos (**mensajes o eventos**) del medio externo.
- En sistemas diseñados como un conjunto de servicios, por lo general se tendrá un disparador y luego **un objeto irá llamando a otro** hasta que todo el sistema se pone en movimiento.

Objeto: "una entidad con comportamiento" (Fontela, 2018, p. 27)

Está definido por:

- *Identidad* (lo que lo distingue de otros objetos de las mismas características)
- *Estado* (los valores de sus **atributos**)
- *Comportamiento* (sus reacciones ante mensajes recibidos y sus acciones en forma de mensajes enviados, implementados mediante **métodos**)



Fuente: Fontela, C. (2018). *Programación Orientada a Objetos*, 3ra. ed.

Ejercicio Nro. 1

Desarrolle un prototipo del juego **Fondo Blanco de Cerveza**.

Al arrancar, dos jugadores se presentan frente a un barril de cerveza lleno (capacidad: 10 litros o 20 dosis de 500 ml).

Cuando sea su turno, cada jugador deberá extraer y beber 1, 2 o 3 dosis de cerveza (según lo que considere que pueda tomar sin parar y lo que el otro jugador le haya dejado en el barril).

El jugador que se sirva la última dosis, pierde.

Paso 1: Encontrar objetos (entidades del dominio del problema)

El desarrollo del software orientado a objetos normalmente comienza con el diseño de un *modelo*.

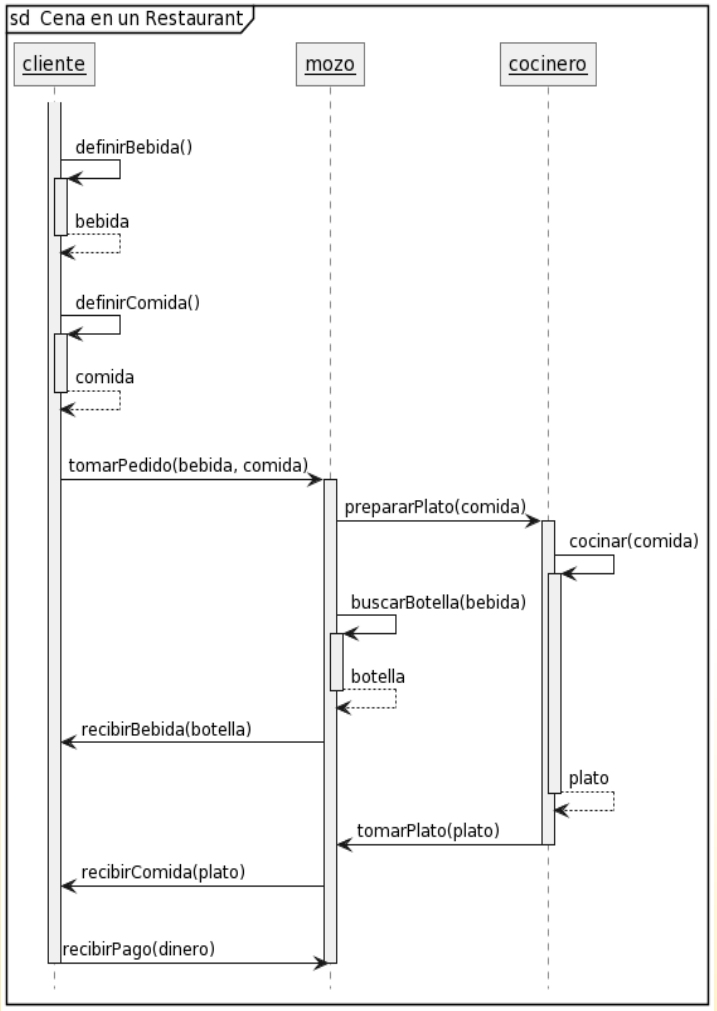
Un **modelo** es una representación de un sistema del mundo real que resulta de llevar a cabo el proceso de *abstracción*.

La **abstracción** es una simplificación que incluye solo aquellos detalles relevantes para determinado propósito y descarta los demás.

Paso 2: Determinar mensajes (cómo deben interactuar los objetos)

UML: Diagramas de secuencia

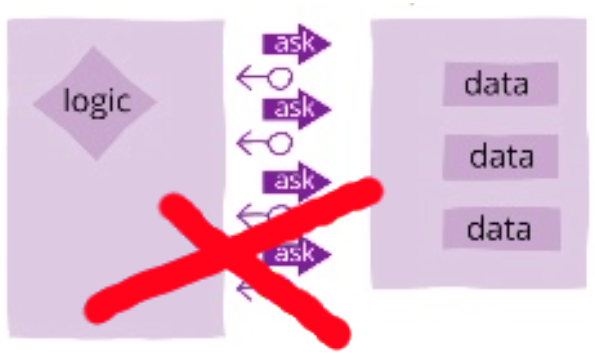
- Los diagramas de secuencia son **modelos dinámicos** que describen cómo un **grupo de objetos** colabora en determinado escenario a través del tiempo.
- En el diagrama de secuencia se representa **cada objeto como un rectángulo** arriba de su **línea de vida**.
- Los **mensajes son flechas** entre objetos y el orden de los mensajes está dado por el eje vertical que se lee de arriba hacia abajo.
- De los mensajes se escribe por lo menos el nombre aunque también se pueden agregar los parámetros.
- También representan cierto control: iteraciones, condiciones, etc.



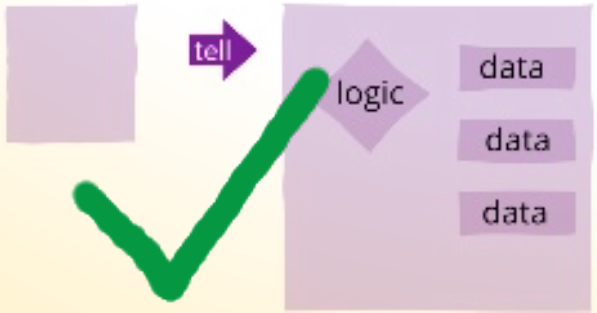
.UBAfiuba
FACULTAD DE INGENIERÍA

Tell, don't ask!

¡Di qué hacer, no preguntes!



Principio de diseño orientado a objetos en el que se basa la **buena práctica** de evitar solicitarle a un objeto que indique su estado y luego llevar a cabo una acción basándose en este, en lugar de solicitarle al objeto que lleve a cabo la acción él mismo.

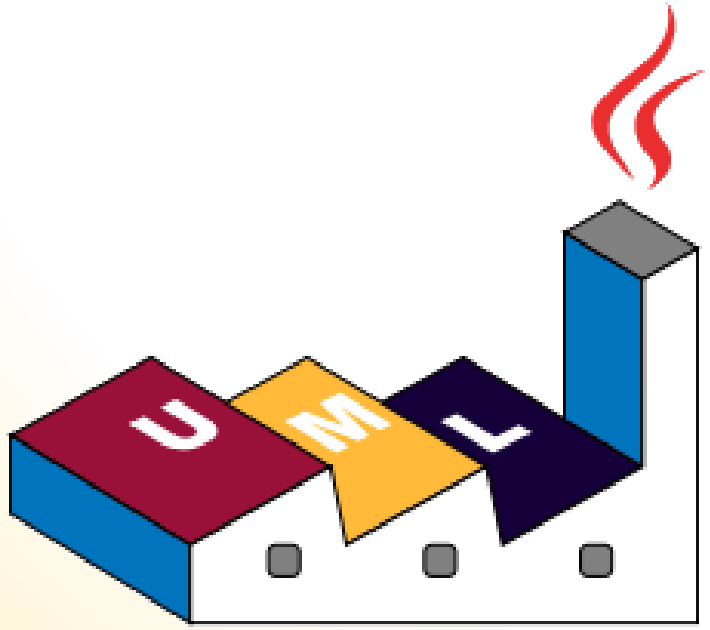


Fuente: https://martinfowler.com/bliki/TellDontAsk.html

PlantUML (https://plantuml.com/es/)

Es un proyecto Open Source que permite generar:

- Diagramas de Secuencia
- Diagramas de Casos de uso
- Diagramas de Clases
- Diagramas de Actividades
- Diagramas de Componentes
- Diagramas de Despliegue, etc.



Los diagramas son definidos usando un lenguaje simple e intuitivo.

Guía de referencia: https://plantuml.com/es/guide

Diagrama de secuencia del juego *Fondo Blanco de Cerveza*

http://www.plantuml.com/plantuml/uml/pLHHIzn047xVNp7O5pquyFQnoA6vQccmu8jzdykEkd
MpaqusOjlhVsysETVaoA5K2Y92Thvllc-oy-
LEwu0IcjAfEsi8RfZ3cXzLVMzzXO8b23VU52oU18u7zP8z5tV29K6GXjG0gkb7GxuW-
Aq0peh0ah4OaXKx5bWxBEuJq3xVTXOz-
E7YEFwry1LAi8Mjq0TeOH9dFOthGzKigexX_TgNzeOkhQTzZ3zAbMZzZM19K1k0SmVV6xfbcCqks1kE
ThvWC0GhaWVwXREPscmsIc5yApXewWR5yW5W3TUg6hpBGITqaQqwB1zZQnInhaEotQvRD2oBlcin2Ue
UrDqKzW43jTwg48fFkRg6-NovXFmZd9pqnUKo5hjIb933e-
VEF7LcAkyxir5dOcRF7eidQhwOv4P68kj4rdl8MaLAPto7FpzHYRL5WzihRwVDsxxvV0-
kb6Ekff1UDSwufZN9ON_jHwtROcjNe2iQX_8L7st9vuz1qAuTHSj5yWnSKT_JYnaghB6oGX-
iGNEMTUAHTJIw0Q7GY8TYonlOKlsbhBYELdtp4oVvJh6sBxfRXdSn6da3SJobCZYPTyDGlCwHmd3Ddz
77C8-
tcyfAeGl3SrfbL4FvPkC7Q6CuoDK4diVaP7D5OcZ79zMJKIea9xtVQPgz6p7zSiJqRiHqB_RQFUg9FU
fNx57_GyJUmMYx4TDJ4NjhmlH_I1XN8HLUoB8te_-t_2p6VcLeyDcFJdfuoaRGy72jJicRfdH_0G00

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

.UBAfiuba
FACULTAD DE INGENIERÍA

```
@startuml
hide footbox
skinparam roundcorner 0
skinparam monochrome true
skinparam sequence {
  ActorBorderColor black
  ActorBackgroundColor #F0F0F0
  Participant underline
  ParticipantBorderColor Black
  ParticipantBackgroundColor #F0F0F0
  LifeLineBackgroundColor #F0F0F0
}

mainframe sd  El juego **Fondo
Blanco de Cerveza**
|||

actor "usuario" as U
participant "juegoFBC" as J
participant "barril" as B
participant "jugador1" as J1
participant "jugador2" as J2
activate U

create J
U -->> J: <<create>>
U -> J: jugar()
activate J

create B
J -->> B: <<create>>
activate B

create J1
J -->> J1: <<create>>
activate J1

create J2
J -->> J2: <<create>>
activate J2

J -> B: hayDemasiadaCerveza()
J <<-- B: hayDemasiadaCerveza

loop hayDemasiadaCerveza
J -> J1: beber(barril)

J1 -> B: calcularMaximoExtraible()
J1 <<-- B: tope

J1 -> J1: decidirCantidadABeber(tope)
activate J1
return cantidad

J1 -> B: esImposibleBeber(cantidad)
J1 <<-- B: esImposibleBeber

loop esImposibleBeber
J1 -> J1: decidirCantidadABeber(tope)
activate J1
return cantidad

J1 -> B: esImposibleBeber(cantidad)
J1 <<-- B: esImposibleBeber
end loop

B <- J1: extraer(cantidad)

J -> B: hayGanador()
J <<-- B: hayGanador

alt hayGanador
J -> J1: cantarVictoria()
else no hayGanador
J -> B: hayPerdedor()
J <<-- B: hayPerdedor

alt hayPerdedor
J -> J2: cantarVictoria()
else no hayPerdedor
J -> J2: beber(barril)

J2 -> B: calcularMaximoExtraible()
J2 <<-- B: tope

J2 -> J2: decidirCantidadABeber(tope)
activate J2
return cantidad

J2 -> B: esImposibleBeber(cantidad)
J2 <<-- B: esImposibleBeber

loop esImposibleBeber
J2 -> J2: decidirCantidadABeber(tope)
activate J2
return cantidad

J2 -> B: esImposibleBeber(cantidad)
J2 <<-- B: esImposibleBeber
end loop

B <- J2: extraer(cantidad)
J -> B: hayGanador()
J <<-- B: hayGanador

alt hayGanador
J -> J2: cantarVictoria()
else no hayGanador
J -> B: hayPerdedor()
J <<-- B: hayPerdedor

opt hayPerdedor
J -> J1: cantarVictoria()
end
end
end
end
J -> B: hayDemasiadaCerveza()
J <<-- B: hayDemasiadaCerveza
end

deactivate J1
deactivate J2
deactivate B
deactivate J
deactivate U
@enduml
```

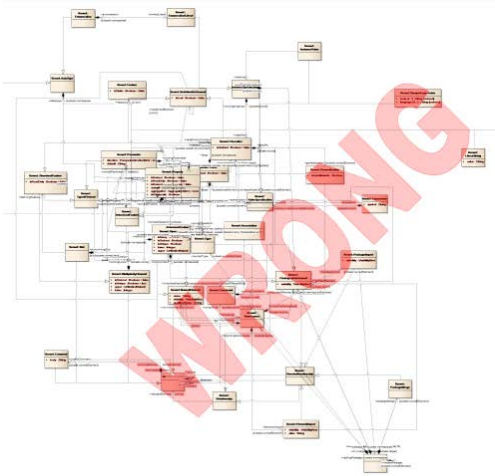
Lectura para reflexionar...

*Sometimes **the tool is talking** a subtle language. It slows down our performance to remind us we need to shape it better. I've lost count of how many times I've been presented **huge UML or entity-relationship diagrams** that require continuous scrolling, impairing my performance in using them. **The diagram was begging the designer to find a reasonable partitioning**, one that could fit on a single, if large, screen. Printing the diagram on even larger paper and working from there is a common way of shutting up our material. It works in the short term, and I use it on early diagrams of large systems, when the architecture isn't necessarily stable. But **I listen, and break large diagrams into subsystems** as soon as I can.*

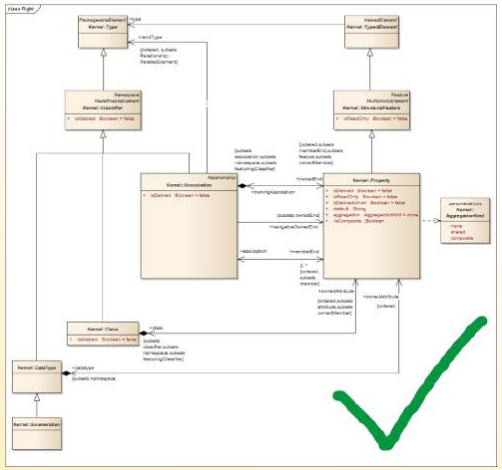
Fuente: Pescio, C. “Listen to your tools and materials”. En: *IEEE Software, sept./oct. 2006*

Less is more

Menos es más



Principio de diseño en el que se basa la **buena práctica** de dividir grandes diagramas en varios diagramas de menor tamaño y mejor enfocados en lo que se quiere comunicar.



Fuente: https://bellekens.com/2012/02/21/uml-best-practice-5-rules-for-better-uml-diagrams/

Ejercicio Nro. 1-A

La solución presentada viola el principio **Menos es más**. Divida el diagrama de secuencia anterior en diagramas de menor tamaño. Siga, para ello, las siguientes *reglas del pulgar*:

"*So, rule of thumb, do a sequence diagram for every basic flow of every use case. Do a sequence diagram for high-level, risky scenarios, and that should be enough. That's* ***how many sequence diagrams*** *I do.*" (Quatrani, 2001, p. 9)

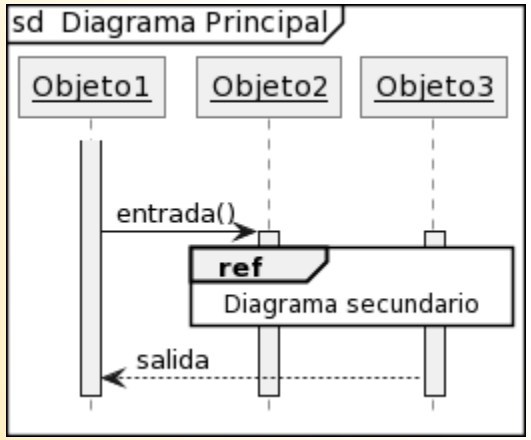
"*My rule of thumb is that you need to be able to print the diagram on a single A4 sheet while keeping things readable.*" (Bellekens, 2012)

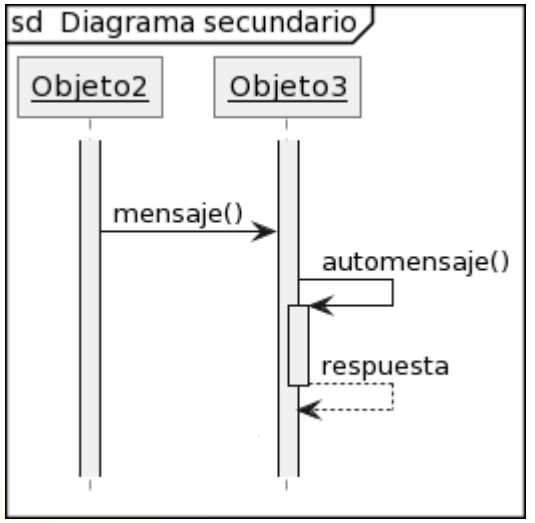
Bellekens, G. (2012). *UML Best Practice: 5 rules for better UML diagrams*.
En: https://bellekens.com/2012/02/21/uml-best-practice-5-rules-for-better-uml-diagrams/

Quatrani, T. "Introduction to the Unified Modeling Language". En: *Rational Developer Network*, 2001

UML: El uso de interacción (*interaction use*)

Permite usar (o llamar) otra interacción, simplificando los **diagramas de secuencia** grandes y complejos. También es común reutilizar alguna interacción entre varias otras interacciones.





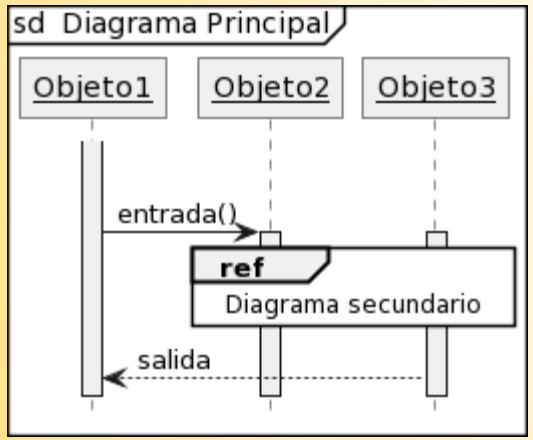
Una restricción impuesta por la especificación UML que a veces es difícil de seguir es que el **fragmento combinado *ref*** debe cubrir todas las *líneas de vida* involucradas representadas en la interacción envolvente. Esto significa que todas esas *líneas de vida* deben ubicarse cerca unas de otras. Si tenemos otro **fragmento combinado *ref*** en el mismo diagrama, podría ser muy complicado reorganizar todas las *líneas de vida* involucradas como lo requiere UML.

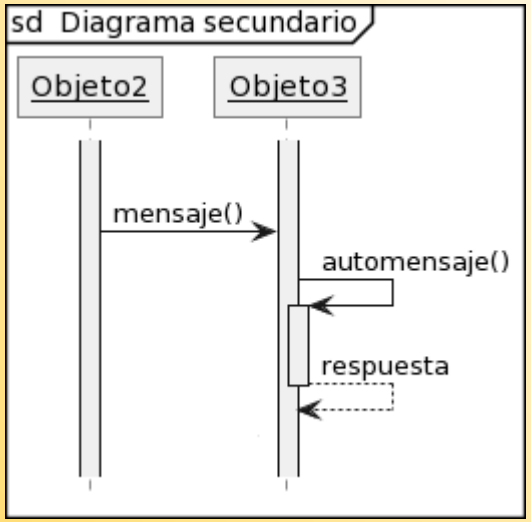
UNIFIED MODELING LANGUAGE™

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

.UBAfiuba
FACULTAD DE INGENIERÍA

```
@startuml
hide footbox
skinparam roundcorner 0
skinparam monochrome true

skinparam sequence {
  Participant underline
  ParticipantBorderColor Black
  ParticipantBackgroundColor #F0F0F0
  LifeLineBackgroundColor #F0F0F0
  ReferenceBorderColor Black
  ReferenceBackgroundColor White
  ReferenceHeaderBackgroundColor #F0F0F0
}

mainframe sd  Diagrama Principal
|||

participant "Objeto1" as O1
participant "Objeto2" as O2
participant "Objeto3" as O3

activate O1
O1 -> O2 : entrada()
activate O2
activate O3
ref over O2, O3 : Diagrama secundario
O3 --> O1 : salida

@enduml
```






```
@startuml
hide footbox
skinparam roundcorner 0
skinparam monochrome true

skinparam sequence {
  Participant underline
  ParticipantBorderColor Black
  ParticipantBackgroundColor #F0F0F0
  LifeLineBackgroundColor #F0F0F0
  ReferenceBorderColor Black
  ReferenceBackgroundColor White
  ReferenceHeaderBackgroundColor #F0F0F0
}

mainframe sd  Diagrama secundario
|||

participant "Objeto2" as O2
participant "Objeto3" as O3

activate O2
activate O3
O2 -> O3: mensaje()
O3 -> O3: automensaje()
activate O3
return respuesta

@enduml
```

UNIFIED MODELING LANGUAGE™

Diagramas de secuencia (conclusión)

Como conclusión podemos decir de los diagramas de secuencia:

- Ayudan a ver el flujo de control y el **ordenamiento temporal de los eventos**
- Ayudan a **encontrar los métodos** necesarios de los objetos
- Conviene colocar más a la izquierda los objetos que inician la **interacción** y los más importantes
- Como todo diagrama, su gran virtud es la **simplicidad** y el impacto visual: no pretenden mostrar toda la complejidad de un sistema.

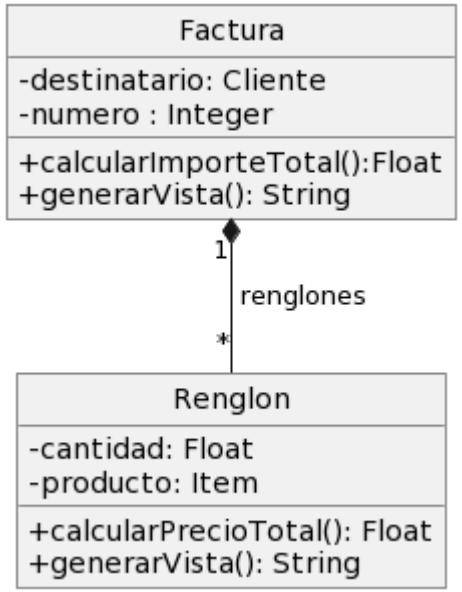
UNIFIED MODELING LANGUAGE™

Paso 3: Implementar el comportamiento de los objetos

- La mayor parte de los lenguajes agrupan los objetos en *clases*, siendo estas **conjuntos de objetos que tienen el mismo comportamiento** (entienden los mismos mensajes y responden de la misma manera; si no fuese así, no diríamos que son de la misma clase).

- Cuando la POO está implementada con *clases*, es **en la definición** de estas que el programador implementa el comportamiento de los objetos que se crearán a partir de ellas.
- En estos lenguajes, todos los objetos son **instancias de clases**.

UML: Diagramas de clases



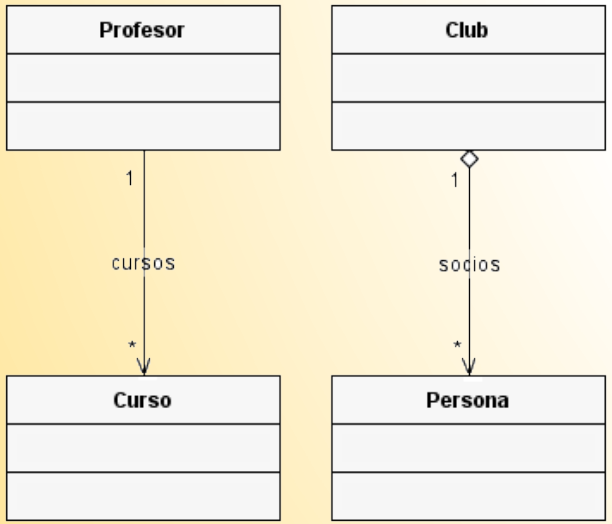
- Representan **relaciones estáticas** entre clases e interfaces (no cambian a través del tiempo)
- Una clase se representa con un **rectángulo con tres divisiones**: una para el *nombre*, otra para los *atributos* y otra para los *métodos*
- En ocasiones no usamos las tres divisiones, sino solo dos (eliminando los atributos) o una (solo con el nombre de la clase)
- Los atributos y métodos **privados** se pueden indicar precedidos de un signo -, los **públicos** del signo + y los **protegidos** del signo #
- No es conveniente utilizar aspectos de la sintaxis que tengan un significado solo en determinado lenguaje

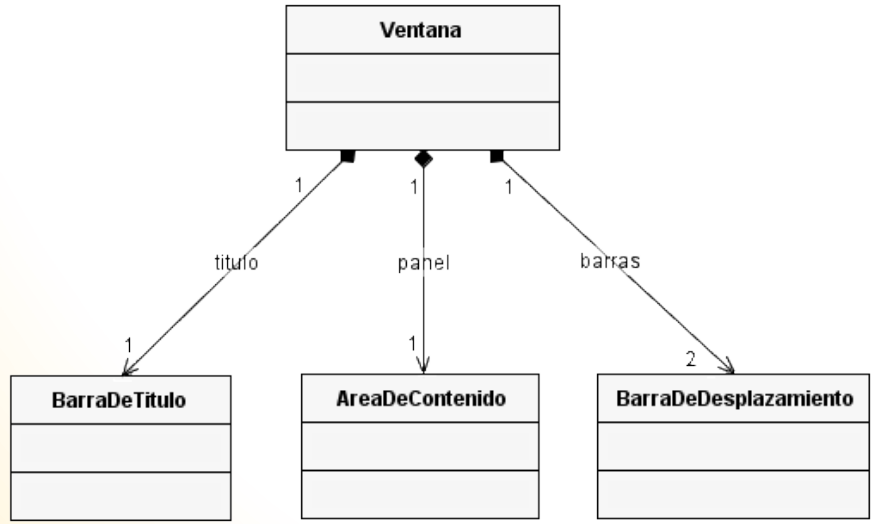
.UBAfiuba
FACULTAD DE INGENIERÍA

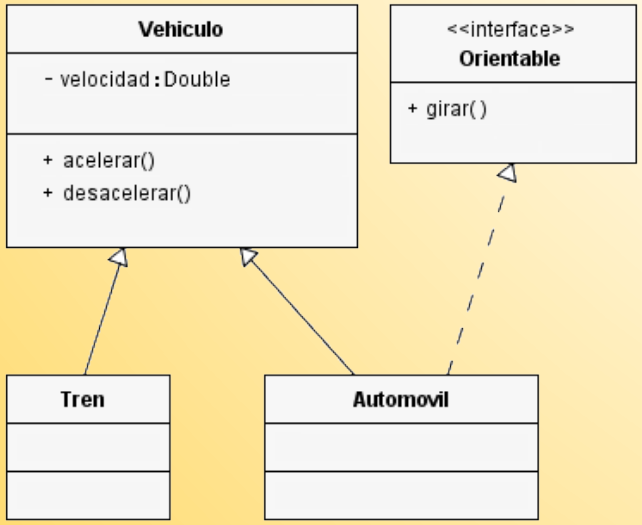
Relaciones entre clases (e interfaces)

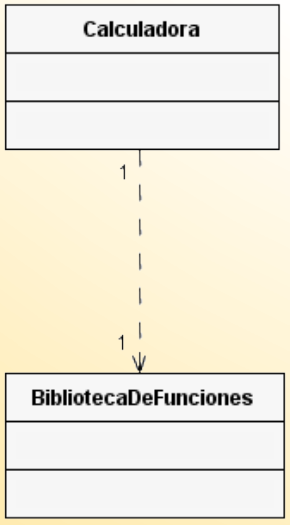
1	uno	***Multiplicidad*** (Número de elementos relacionados)
0..1	cero o uno	
0..*	cero o muchos	
1..*	uno o muchos	
*	cero o muchos	

UNIFIED MODELING LANGUAGE™









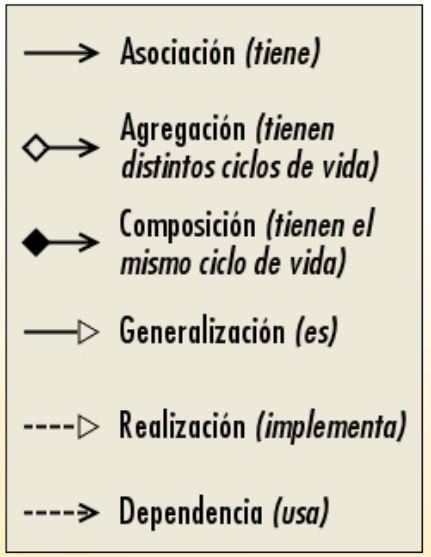
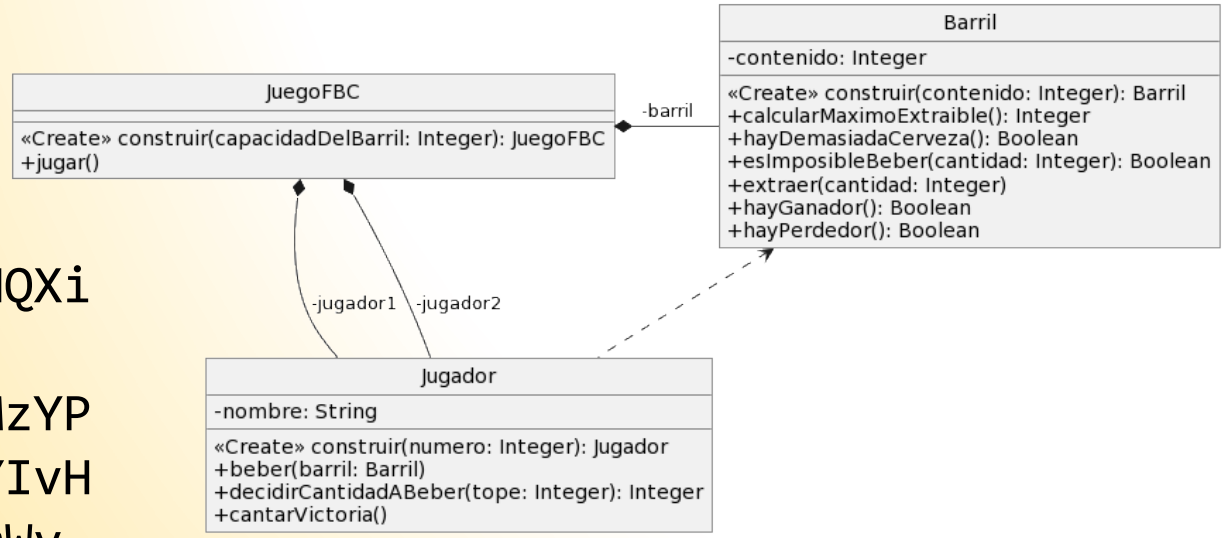


Diagrama de clases del juego *Fondo Blanco de Cerveza*



http://www.plantuml.com/plantuml/uml/XL7HQXin47pNL-WnQN5e-sXCI7nfYmk5Gg3lUz9otbJI7Ykff2dzzqeddn6j8MzYPsPdTdGN4qZAtgaZMTI6n3XKyGU56GIyzXpO78KzwYIvHuo36EzJ4XfpmeFXy4WlgDzr5E4Sh649A6Mk5eNUWmWv_LjjTD4a36Hvgm_bDQ6etMuGX8ItjnMDnPFawd_UzVQqHxtL1fp93kGhF9FdZyz9W4Q7LzTdTY4TuTS3UeW45WQKd_W2bR1dTWYX430U_ConAlSuOZ65aCY2xJnxUhMvo6fcdo60PUayslmRYiL_W3zhDL_oL4LBDu7zABZLZqaeJ9Tx2TcZzAMSDnIpSRbYN5fQswhXB1go9CCf-7sxDl6CtQAkkNeXo7SoYOLAQNrSdFZJVYXvBtyRp62gnmEwvjyxhEgQzQd4bhRwF7wpqQjeqwxeiSrwQa6VsklzA_W7rVRftStDYgex33PxznS0

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

.UBAfiuba
FACULTAD DE INGENIERÍA

```
@startuml
hide circle
skinparam monochrome true
skinparam classAttributeIconSize 0
skinparam roundcorner 0

class Barril {
- contenido: Integer
<<Create>> construir(contenido: Integer): Barril
+ calcularMaximoExtraible(): Integer
+ hayDemasiadaCerveza(): Boolean
+ esImposibleBeber(cantidad: Integer): Boolean
+ extraer(cantidad: Integer)
+ hayGanador(): Boolean
+ hayPerdedor(): Boolean
}

class Jugador {
- nombre: String
<<Create>> construir(numero: Integer): Jugador
+ beber(barril: Barril)
+ decidirCantidadABeber(tope: Integer): Integer
+ cantarVictoria()
}

class JuegoFBC {
<<Create>> construir(capacidadDelBarril: Integer): JuegoFBC
+ jugar()
}

JuegoFBC *- Barril: -barril
JuegoFBC *-- Jugador: -jugador1
JuegoFBC *-- Jugador: -jugador2
Barril <.. Jugador
@enduml
```

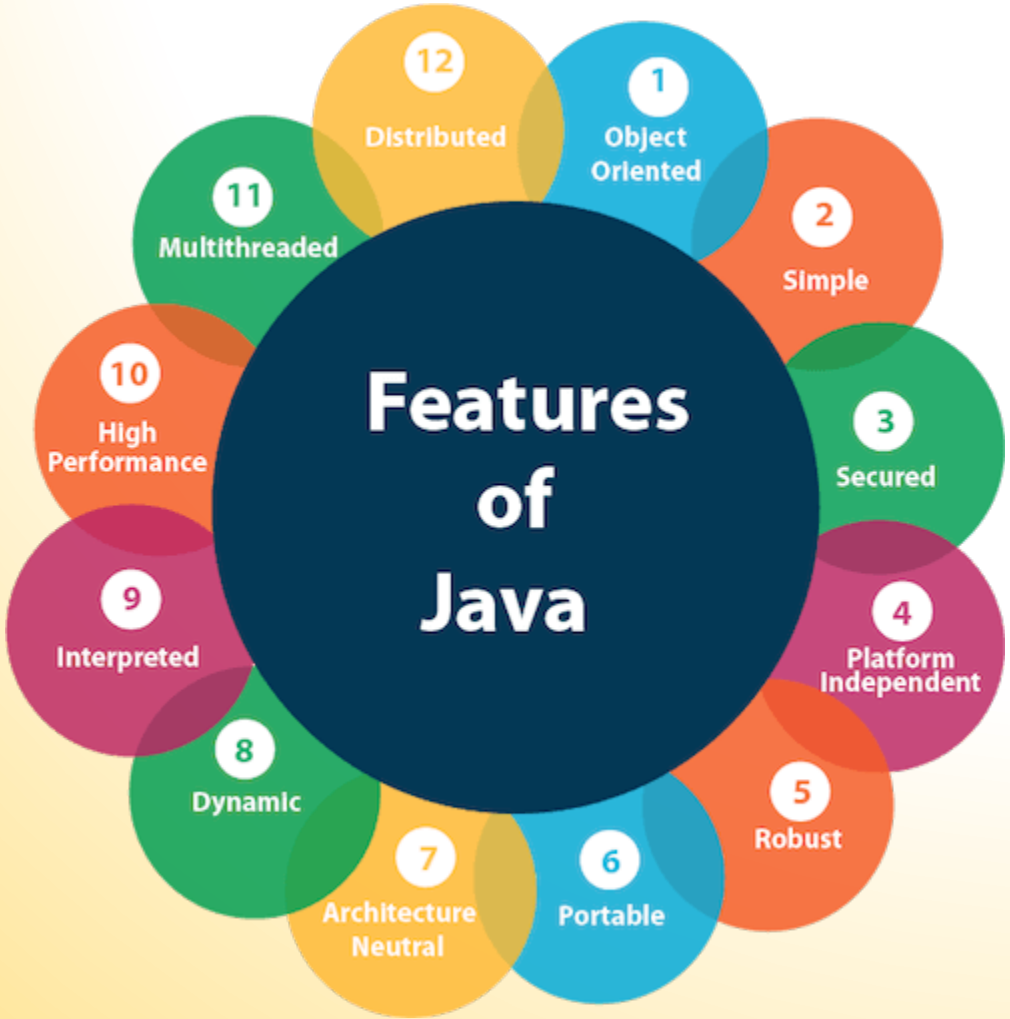
Diagramas de clases (conclusión)

- No es necesario representar todas las clases con todos sus detalles
- Conviene representar atributos, asociaciones y generalización (la visibilidad, las dependencias y otras propiedades son necesarias solo en algunas ocasiones especiales)
- Conviene dibujar modelos solo de las partes importantes del sistema
- Son herramientas, por lo que el grado de detalle hay que manejarlo desde el punto de vista de la practicidad
- Conviene ordenar los elementos del diagrama para minimizar el cruce de líneas y para que las cosas que son cercanas semánticamente queden cerca físicamente

UNIFIED MODELING LANGUAGE™

Características de Java

1. Orientado a objetos
2. Simple
3. Seguro
4. Independiente de la plataforma
5. Robusto
6. Portable
7. Arquitectónicamente neutro
8. Dinámico
9. Interpretado
10. Alto desempeño
11. Multihilo
12. Distribuido



Fuente: https://www.javatpoint.com/features-of-java

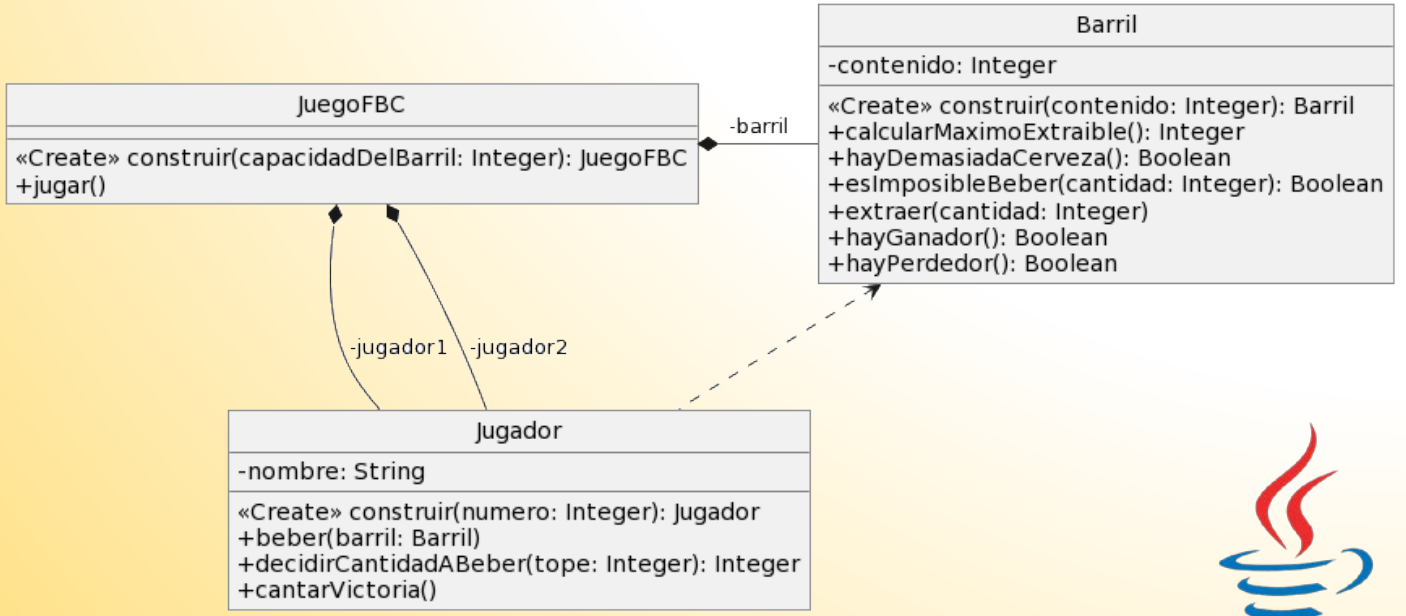
Implementación del juego *Fondo Blanco de Cerveza*

El archivo Main.java

```
1  package juegoFBC;
2
3  public class Main {
4      public static final java.util.Scanner teclado = new java.util.Scanner(System.in);
5      public static final java.io.PrintStream pantalla = new java.io.PrintStream(System.out);
6
7      public static void main(String[] args) {
8          JuegoFBC juegoFBC = new JuegoFBC(20);
9          juegoFBC.jugar();
10     }
11
12 }
```

```java
package juegoFBC;

public class JuegoFBC {

    private Barril barril;
    private Jugador jugador1;
    private Jugador jugador2;

    public JuegoFBC(int capacidadDelBarril) {
        barril = new Barril(capacidadDelBarril);
        jugador1 = new Jugador(1);
        jugador2 = new Jugador(2);
    }

    public void jugar() {
        while (barril.hayDemasiadaCerveza()) {
            jugador1.beber(barril);
            if (barril.hayGanador()) {
                jugador1.cantarVictoria();
            } else if (barril.hayPerdedor()) {
                jugador2.cantarVictoria();
            } else {
                jugador2.beber(barril);
                if (barril.hayGanador()) {
                    jugador2.cantarVictoria();
                } else if (barril.hayPerdedor()) {
                    jugador1.cantarVictoria();
                }
            }
        }
    }

}
```

Implementación del juego *Fondo Blanco de Cerveza*

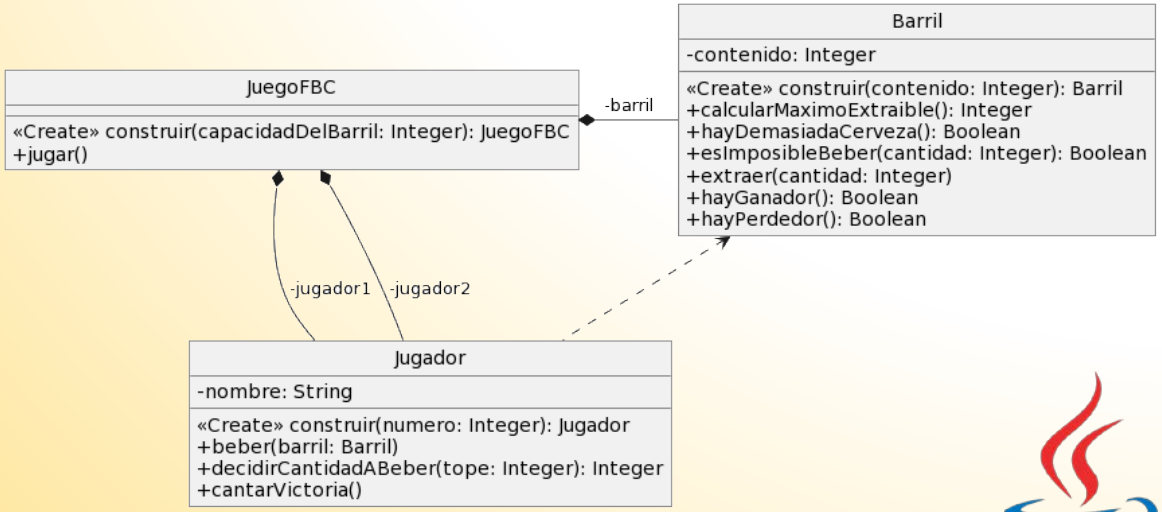


El archivo JuegoFBC.java

Java

```java
package juegoFBC;

public class Barril {

    private int contenido;

    public Barril(int contenido) {
        this.contenido = contenido;
    }

    public boolean hayDemasiadaCerveza() {
        return contenido > 1;
    }

    public boolean hayGanador() {
        return contenido == 1;
    }

    public boolean hayPerdedor() {
        return contenido == 0;
    }

    public boolean esImposibleBeber(int cantidad) {
        return cantidad < 1 || cantidad > 3 || cantidad > contenido;
    }

    public int calcularMaximoExtraible() {
        return (contenido > 2 ? 3 : contenido);
    }

    public void extraer(int cantidad) {
        contenido -= cantidad;
    }

}
```

Implementación del juego *Fondo Blanco de Cerveza*

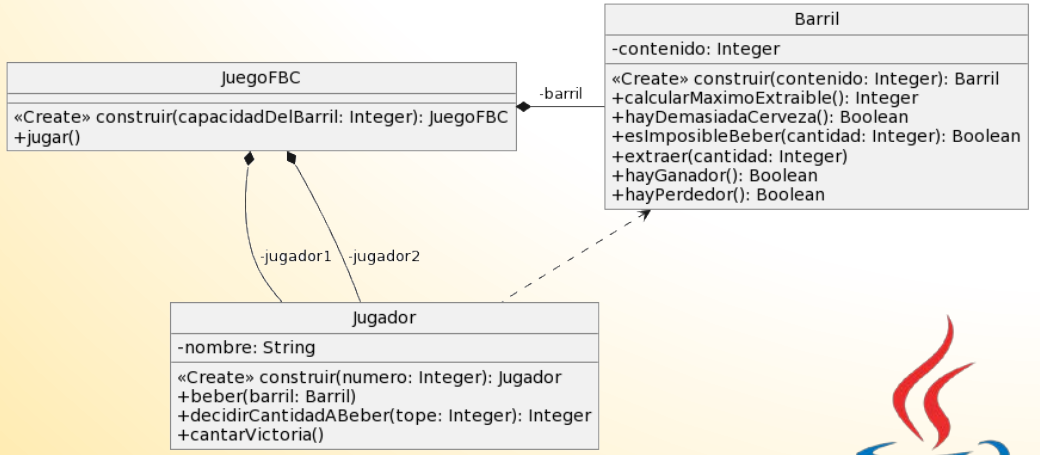


El archivo Barril.java

Java

```java
package juegoFBC;

import static juegoFBC.Main.pantalla;
import static juegoFBC.Main.teclado;

public class Jugador {

    private String nombre;

    public Jugador(int numero) {
        do {
            pantalla.print("Jugador Nro. " + numero + ", ingrese su nombre: ");
            nombre = teclado.nextLine();
        } while (nombre.isBlank());
    }

    public void beber(Barril barril) {
        int tope = barril.calcularMaximoExtraible();
        int cantidad = decidirCantidadABeber(tope);
        while (barril.esImposibleBeber(cantidad)) {
            cantidad = decidirCantidadABeber(tope);
        }
        barril.extraer(cantidad);
    }

    private int decidirCantidadABeber(int tope) {
        pantalla.print(nombre + ", cuantas dosis de 500 ml vas a beber? (max. " + tope + "): ");
        String renglon = teclado.nextLine();
        int cantidad;
        try {
            cantidad = Integer.parseInt(renglon);
        } catch (Exception ex) {
            cantidad = 0;
        }
        return cantidad;
    }

    public void cantarVictoria() {
        pantalla.println(nombre + " es el vencedor! Vayan a comprar otro barril...");
    }
}
```

Implementación del juego *Fondo Blanco de Cerveza*



El archivo Jugador.java

Ejecución del juego *Fondo Blanco de Cerveza*

```
Jugador Nro. 1, ingrese su nombre: Sigfrid
Jugador Nro. 2, ingrese su nombre: Hans
Sigfrid, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Hans, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Sigfrid, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Hans, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Sigfrid, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Hans, cuantas dosis de 500 ml vas a beber? (max. 3): 3
Sigfrid, cuantas dosis de 500 ml vas a beber? (max. 2): 1
Sigfrid es el vencedor! Vayan a comprar otro barril...
```

Estado de los objetos

- Los objetos que interactúan en un sistema orientado a objetos tienen un **estado** que está dado por los valores de sus **atributos** (*variables de instancia*) en determinado momento.
- Generalmente, el estado de un objeto evoluciona en el tiempo, ya que desde sus métodos es posible acceder a sus atributos y modificarles el valor. La excepción son los atributos calificados como `final`, ya que estos permanecen *constantes*.
- Por convención, en Java los nombres de los atributos empiezan en minúsculas. La excepción son los atributos calificados como `final`, que se escriben totalmente en mayúsculas.
- Los atributos de un objeto no deberían ser manipulables directamente por el resto de los objetos del sistema, por eso casi siempre se los califica como `private` y se accede a ellos mediante métodos.
- Se recomienda *declarar* los atributos (es decir, indicar su tipo y su nombre) antes de los constructores, y utilizar éstos para *inicializarlos* (es decir, asignarles su primer valor).

Tipos de datos de los atributos: Primitivos

- En Java, cualquier variable declarada usando un *tipo primitivo* contendrá directamente un dato.

Tipo	Rango	Valor por defecto
byte	-128 .. 127	0
short	-32768 .. 32767	0
int	-2147483648 .. 2147483647	0
long	-9223372036854775808 .. 9223372036854775807	0L
float	-3.4E38 .. -1.18E-38 .. 0 .. 1.18E-38 .. 3.4E38	0.0f
double	-1.8E308 .. -2.23E-308 .. 0 .. 2.23E-308 .. 1.8E308	0.0d
char	'\u0000' .. '\uffff'	'\u0000'
boolean	false .. true	false

- En cambio, si el tipo usado para declarar una variable no es primitivo, esta contendrá una *referencia a un objeto*, es decir, la dirección de memoria donde el objeto está almacenado.
- El *valor por defecto* es el valor que toma una *variable de instancia* (es decir, un atributo) cuando se la usa **sin inicialización** previa. En el caso de las *variables locales* de los constructores y los métodos, su uso sin inicialización previa no es posible (no compila).

Tipos de datos de los atributos: String

- Un atributo que represente el texto característico de un objeto (por ejemplo, el apellido de una persona), puede implementarse en Java como una instancia de la clase `String`.
- Esta clase tiene 13 constructores diferentes que permiten construir cadenas de caracteres. Por ejemplo, si escribiéramos lo siguiente: `new String (new char[]{'h', 'o', 'l', 'a'})` estaríamos creando una instancia de `String` a partir de un arreglo *anónimo* de 4 caracteres.
- Por una cuestión de practicidad, en Java es posible crear un objeto de la clase `String` simplemente colocando caracteres entre comillas: `"Hola, mundo!"`
- El valor por defecto de los atributos que son instancias de la clase `String` (o de cualquier otra clase) es `null`. Cuando una variable vale `null`, esta no se refiere a ningún objeto, por ello no es posible enviarle mensajes. La cadena vacía `""` sí es un objeto. Por lo tanto, una variable `String` inicializada con la cadena vacía puede responder a los mensajes que le enviemos (`length`, `equals`, `charAt`, `substring`, `concat`, `contains`, `indexOf`, `isEmpty`, `trim`, `startsWith`, `endsWith`, `toLowerCase`, `toUpperCase`, `replace`, `replaceAll`, entre otros).

Tipos de datos de los atributos: Arreglos y colecciones

- Un atributo que represente un grupo de elementos del mismo tipo primitivo (por ejemplo, los números premiados en un sorteo) o un grupo de instancias de la misma clase (por ejemplo, los alumnos inscriptos en un curso) puede implementarse en Java como un *arreglo* o una *colección*.
- En UML, para indicar cualquiera de las dos implementaciones se utiliza la **multiplicidad**.
- Un *arreglo* es un objeto contenedor cuyo tamaño se define al momento de su instanciación y que no acepta demasiados mensajes (`copyOfRange`, `equals`, `fill`, `binarySearch` y `sort`, entre otros). El acceso a sus posiciones se realiza mediante un índice entre [ ], y se les cargan valores con =.
- Las *colecciones* no tienen un tamaño fijo (se adaptan a medida que se les va cargando su contenido). Entre las más usadas se encuentran `ArrayList`, `LinkedList` y `TreeSet`. Aceptan muchos mensajes distintos, como `isEmpty`, `size`, `add`, `set`, `get`, `remove`, `indexOf`, `contains`, etc. *Solo pueden contener datos primitivos si estos se envuelven en objetos*.

Arreglos: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
Colecciones: https://docs.oracle.com/javase/tutorial/collections/TOC.html

Comportamiento de los objetos

- Todo objeto tiene un **comportamiento** que está dado por las funciones (**métodos**) que ejecuta cuando recibe solicitudes (**mensajes**), generalmente de otros objetos.
- También durante la instanciación de los objetos se ejecutan instrucciones, por ejemplo, para inicializar sus atributos. Este comportamiento está codificado en **constructores**, los cuales no tienen valor de retorno (ni siquiera `void`) y deben tener el mismo nombre que su clase.
- Cada método debe limitarse a realizar **una única tarea bien definida**, y su nombre debe expresar esa tarea con efectividad. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.
- Desde el resto de los objetos se debería poder solicitarle a un objeto la ejecución de sus métodos, por eso a estos casi siempre se los califica como `public`. Una excepción son aquellos métodos que solo se invocarán desde dentro del propio objeto, en cuyo caso se los debe hacer invisibles para los demás objetos, calificándolos como `private`.

Comportamiento de los objetos

- Hay tres formas de *invocar un método*:
  1. Pasándole un mensaje a un objeto, es decir, usando una variable que se refiera al objeto, seguida de punto (`.`) y el nombre del método. Por ejemplo: `estudiante.listar();`
  2. Utilizando el nombre de la clase, seguido de punto (`.`) y el nombre de un método de la clase. Por ejemplo: `Math.sqrt(81.0)`
  3. Usando solo el nombre del método, si este es parte de la misma clase. Por ejemplo: `cerrar()`

- Existen tres formas *de finalizar un método* y regresar el control al código que lo invocó. Si el método no devuelve un resultado, el control regresa cuando:
  1. el flujo del programa llega a la llave de cierre del método,
  2. se ejecuta la sentencia `return;`

  Si el método devuelve un resultado, el control regresa cuando:
  3. se ejecuta la sentencia `return` *expresión*`;` la cual evalúa la expresión y después devuelve el resultado al código que hizo la invocación.

Comportamiento de los objetos

- Los métodos pueden devolver como máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga varios valores. El tipo del valor devuelto se indica antes del nombre del método, al declararse este último. La palabra reservada `void` se utiliza para indicar que un método no devuelve ningún valor.
- Los métodos (y también los constructores) pueden tener *variables locales*. Solamente es posible acceder a las variables locales dentro del ámbito en que están declaradas, y al finalizar la ejecución del método o constructor al que pertenecen, el valor de las mismas no se mantiene.
- Los métodos (pero no los constructores) pueden ser *recursivos*.
- Un método (o un constructor) puede ser invocado con cero o más **argumentos**, si hay una declaración que tenga una firma compatible. La firma (*signature*) de un método o constructor está compuesta por su nombre y los tipos de sus **parámetros**, por lo tanto es posible que haya múltiples versiones de un método o constructor, siempre y cuando sus firmas difieran. Esto se denomina *sobrecarga*.

Comportamiento de los objetos

- Una característica importante de las invocaciones de los métodos es la **promoción de argumentos**. Por ejemplo, se puede llamar al método estático `sqrt` de la clase `Math` con un argumento entero, a pesar de que el método espera recibir un `double`.
- Tratar de realizar estas conversiones puede ocasionar errores de compilación, si no se satisfacen las reglas de promoción que especifican qué conversiones son permitidas y pueden realizarse sin perder datos. En casos en los que la información podría perderse debido a la conversión, el compilador requerirá que utilicemos un *casteo* para forzar explícitamente la conversión.
- En Java, el pasaje de argumentos a los métodos es **por valor**. Lo que se le pasa al método invocado es una copia del valor del argumento. Si el argumento es una variable, las modificaciones a la copia no afectan el valor de la variable original. Por ejemplo: `borrar(x)` no cambia el valor de la variable x. Sin embargo, si el parámetro no es de un tipo primitivo, el método que lo recibe puede (a través de la copia) acceder a los miembros públicos del objeto e interactuar con él, incluso modificándole el estado.

Ejercicio Nro. 1-B

Desarrolle una versión *multijugador* (n > 2) del juego ***Fondo Blanco de Cerveza.***

```
package juegoFBC;
public class JuegoFBC {
 private Barril barril;
 private Jugador jugador1;
 private Jugador jugador2;
 public JuegoFBC(int capacidadDelBarril) {
  barril = new Barril(capacidadDelBarril);
  jugador1 = new Jugador(1);
  jugador2 = new Jugador(2);
 }
 public void jugar() {
  while (barril.hayDemasiadaCerveza()) {
   jugador1.beber(barril);
   if (barril.hayGanador()) {
    jugador1.cantarVictoria();
   } else if (barril.hayPerdedor()) {
    jugador2.cantarVictoria();
   } else {
    jugador2.beber(barril);
    if (barril.hayGanador()) {
     jugador2.cantarVictoria();
    } else if (barril.hayPerdedor()) {
     jugador1.cantarVictoria();
    }
   }
  }
 }
}
```

```
package juegoFBC;
public class Barril {
 private int contenido;
 public Barril(int contenido) {
  this.contenido = contenido;
 }
 public boolean hayDemasiadaCerveza() {
  return contenido > 1;
 }
 public boolean hayGanador() {
  return contenido == 1;
 }
 public boolean hayPerdedor() {
  return contenido == 0;
 }
 public boolean esImposibleBeber(int cantidad) {
  return cantidad < 1 || cantidad > 3 || cantidad
> contenido;
 }
 public int calcularMaximoExtraible() {
  return (contenido > 2 ? 3 : contenido);
 }
 public void extraer(int cantidad) {
  contenido -= cantidad;
 }
}
```

```
package juegoFBC;
import static juegoFBC.Main.pantalla;
import static juegoFBC.Main.teclado;
public class Jugador {
 private String nombre;
 public Jugador(int numero) {
  do {
   pantalla.print("Jugador Nro. " + numero + ", ingrese su nombre: ");
   nombre = teclado.nextLine();
  } while (nombre.isBlank());
 }
 public void beber(Barril barril) {
  int tope = barril.calcularMaximoExtraible();
  int cantidad = decidirCantidadABeber(tope);
  while (barril.esImposibleBeber(cantidad)) {
   cantidad = decidirCantidadABeber(tope);
  }
  barril.extraer(cantidad);
 }
 private int decidirCantidadABeber(int tope) {
  pantalla.print(nombre + ", cuantas dosis de 500 ml vas a beber? (max. " + tope + "): ");
  String renglon = teclado.nextLine();
  int cantidad;
  try {
   cantidad = Integer.parseInt(renglon);
  } catch (Exception ex) {
   cantidad = 0;
  }
  return cantidad;
 }
 public void cantarVictoria() {
  pantalla.println(nombre + " es el vencedor! Vayan a comprar otro barril...");
 }
}
```

```
package juegoFBC;
public class Main {
 public static final java.util.Scanner teclado = new java.util.Scanner(System.in);
 public static final java.io.PrintStream pantalla = new java.io.PrintStream(System.out);
 public static void main(String[] args) {
  JuegoFBC juegoFBC = new JuegoFBC(20);
  juegoFBC.jugar();
 }
}
```