Primeras guías para comunicar un diseño

Por
Fernando Dodino
Carlos Lombardi

Versión 3.2
Diciembre 2016

Indice

1 Objetivo
- 1.1 Código
- 1.2 UML (Unified Modeling Language)
- 1.3 Diagramas estáticos y dinámicos
- 1.4 Un enunciado

2 Diagrama de Clases
- 2.1 Antes que nada: un socio no es un socio…
- 2.2 Tips para documentar clases
- 2.3 Relaciones entre clases
  - 2.3.1 RELACION “HEREDA”
  - 2.3.2 RELACIÓN “CONOCE”
  - 2.3.3 RELACIÓN “IMPLEMENTA”
  - 2.3.4 RELACIÓN “USA”
- 2.4 Tips para documentar un diagrama de clases
- 2.5 Un breve ejemplo en Wollok

3 Diagrama de Objetos
- 3.1 Objeto
- 3.2 Ambiente
- 3.3 Referencias
- 3.4 Tips para documentar un diagrama de objetos

4 Diagrama de Secuencia (BONUS)
- 4.1 Actores/Objetos
- 4.2 Comunicación
- 4.3 Creación de objetos
- 4.4 Tips para documentar un diagrama de secuencia

5 Resumen

6 Bibliografía complementaria

1 Objetivo

El presente apunte tiene como objetivo ofrecer una guía para los que necesiten comunicar el diseño de un trabajo práctico, examen, ejercicio, o bien para su uso en el campo profesional. Para ello tomamos en cuenta que no existe una única verdad pero sí buenas prácticas para documentar una solución; esas prácticas facilitarán la comprensión a quien esté del otro lado (sean docentes o compañeros).

¿En qué lenguaje hablamos cuando estamos diseñando? En general, vamos a utilizar:

1.1 Código

Es un buen momento para recordar que el código es una forma totalmente válida para evitar ambigüedades cuando describimos nuestra solución. Bajar a detalle las partes más relevantes nos va a permitir darnos cuenta de que

- determinado objeto necesite recibir más información o conocer a ciertos objetos
- un objeto deba delegar responsabilidades a otro objeto
- aparezcan nuevas abstracciones que no habíamos pensado originalmente

etc.

1.2 UML (Unified Modeling Language)

Es un **lenguaje** o herramienta de comunicación para las personas que permite representar un modelo (una idea) del sistema real. Recordemos que un modelo es una abstracción que expresamos a través de los distintos diagramas: *el modelo no es el sistema, sino lo que me interesa representar del sistema*.

Autores: Booch, Rumbaugh y Jacobson, quienes habían creado cada uno por su lado una herramienta para modelar con objetos (por eso lo de Unificado).

- **sintaxis del UML:** los distintos diagramas, del que nos importarán específicamente el diagrama de clases y el de objetos. Como material extra aparece el diagrama de secuencia.

- **semántica del UML:** el paradigma orientado a objetos.

Como herramienta de comunicación, **el objetivo principal de los diagramas UML es la comprensión del lector antes que su completitud**. Entonces:

- **Sólo lo que nosotros queremos comunicar debe mostrarse en un diagrama**
- Expresamos una abstracción del problema, esto significa perder información que no necesito y quedarme sólo con lo esencial para comprender lo que estoy mostrando. Así puedo tratar la complejidad de ese problema.
- Adoptar como estrategia mostrar sólo lo importante, en lugar de documentar todo, nos permitirá que los diagramas sean claros y simples, evitando quedar enmarañados entre flechas y rectángulos.

1.3 Diagramas estáticos y dinámicos

UML nos brinda varios diagramas o puntos de vista para describir una solución:

- Algunos diagramas se centran en los aspectos estáticos: las clases, sus relaciones y el código, que no dependen de casos puntuales (son generales)
- Otros (los dinámicos) trabajan a través de ejemplos concretos: el alumno Nicolás, la factura 0001-00070781, etc.

Ambos tipos de diagrama no son excluyentes sino que por el contrario se complementan y facilitan entender la forma de resolver un problema desde la visión general y desde la casuística particular.

A continuación describimos los diagramas más usados en la materia, tomando en cuenta que el lector ya tiene conocimientos profundos de la teoría de objetos[1].

1.4 Un enunciado

Se quiere modelar los socios de una biblioteca. Un socio se registra en la biblioteca proveyendo su nombre y apellido. Entonces se define el monto que tiene que pagar como cuota mensual según el siguiente criterio:

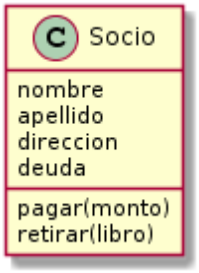
- Los socios vitalicios no pagan cuota
- A los socios comunes se les cobra $ 20 y $ 10 adicionales si llevaron más de 3 libros en el mes anterior
- A los socios especiales se les cobra $ 25 sin límite de préstamo de libros

El socio puede pagar en cualquier momento la cuota de un mes determinado, incluso puede pagar menos del monto de la cuota. **Ejemplo**: Felipe es socio especial (se le cobra $ 25), decide pagar $ 10 de la cuota de noviembre. Si no tiene pagos anteriores le queda como saldo $ 15.

Cada vez que un socio quiere retirar un libro, se verifica que no deba más del monto de una cuota, en caso contrario se avisa que debe pagar para poder concretar el préstamo.

Ahora veremos cómo documentar una solución posible para el presente requerimiento…

2 Diagrama de Clases



El diagrama de clases es una herramienta que permite modelar las relaciones entre las entidades. En UML, una clase es representada por un rectángulo que posee tres divisiones:

En el primer cuadro anotamos el nombre de la clase (si es abstracta se escribe en cursiva, o bien se usa un estereotipo <<abstract>> arriba del nombre de la clase)

[1] Puede verse al respecto los apuntes de Teoría de Objetos de la cátedra.

En la segunda parte (que es opcional, según veremos en los tips que están a continuación) van los atributos (o variables de instancia). La sintaxis para especificar atributos que vamos a utilizar es: ***nombre***

Si bien UML permite especificar tipos para los atributos, e incluso niveles de visibilidad de la variable (si es privada o pública, entre otros) nuestro consejo es comunicar el nombre que es el único dato obligatorio.

En el último cuadro escribimos las **operaciones** (**qué** mensajes que puede entender). No confundir con los **métodos** que es **cómo** lo resuelve cada objeto. La diferencia es similar a *interfaz* (operaciones) vs. *implementación* (el método, la porción de código que se ejecuta cuando envío un mensaje a un objeto).
La sintaxis para especificar operaciones es:

```
nombre (opcional: parámetros)
```

Solo el nombre es obligatorio, conviene leer las recomendaciones que más adelante haremos.

2.1 Antes que nada: un socio no es un socio…

Antes de seguir repasemos la definición de la clase Socio:

El socio real vs. el objeto socio que vive en un ambiente

Cuando uno empieza a diseñar, es frecuente pensar: “A mí me parece mal que el Socio tenga un atributo deuda, porque en la realidad no es el socio el que sabe su deuda, es la Biblioteca la que registra cuánta plata debe el socio…”

Se suele decir que “objetos nos permite modelar fácilmente la realidad”. Claro, esta frase cierra si recordamos lo dicho anteriormente: el modelo **no** es la

realidad y nuestro objetivo no es reflejar la realidad tal cual, sino tener una simplificación de esa realidad infinitamente compleja que resuelva un problema particular (requerimiento). También lo llamamos socio, pero no es el socio "verdadero", es una representación de aquel socio.

Entonces nos damos cuenta de que para armar una aplicación nos conviene mucho más que la deuda la maneje el objeto socio y que no sea una responsabilidad más de la biblioteca.

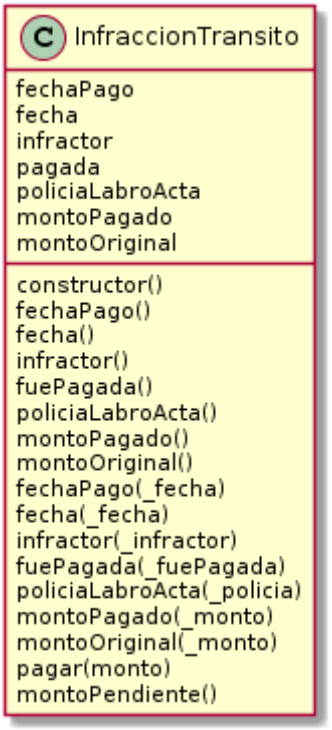
2.2 Tips para documentar clases

Lo que más nos interesa de una clase es qué es lo que le podemos pedir, entonces:

- Conviene tomarse un tiempo para ponerles buenos nombres a nuestras abstracciones (sean clases, métodos, etc.). Y cuando el nombre no refleja la abstracción que nosotros originalmente pensamos, vale cambiarlo (en el papel cuesta un poco más que en la máquina, pero vale la pena).

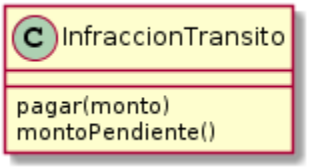
- Salvo que sea una decisión de diseño importante, no es recomendable documentar las variables. Esto tiene que ver con el punto de vista desde el cual nos queremos parar frente a un objeto: al armar el diagrama de clases pienso primero en los servicios que ofrece cada clase y no en cómo implementar esos servicios. *Ejemplo*: si tengo una clase Jugador y tengo un mensaje cantidadGoles(), es preferible no pensar de antemano si voy a tener un atributo cantidadGoles entero o bien lo voy a obtener de la colección de los partidos disputados. Eso me da más libertad para modificar mi implementación sin cambiar la interfaz de los objetos que usan al Jugador. Pensar en la estructura de un objeto me ata más a cómo lo tengo que implementar, algo que en todo caso eso puede resolverse más adelante con un poco de código.

- Como dijimos antes lo importante no es documentar todos los mensajes de un objeto, sino sólo los más relevantes. Así quedan fuera los getters, setters, métodos privados y aquellos que uno considere *menores*. Para prueba basta el siguiente ejemplo:



Una clase con demasiada información

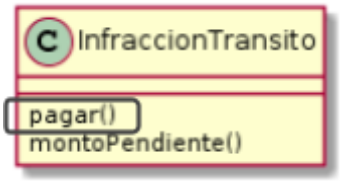
¿Cómo sabemos qué métodos son importantes? Tenemos que ir al código para estar seguros: entonces el diagrama de clases tiene tanta información que no sirve para nada. La clase bien podría documentarse de esta manera:



Ahora queda mucho más claro qué me interesa pedirle a una infracción de tránsito. **Moraleja**: cuidado con las herramientas que, en base al código, generan el diagrama (y viceversa). Bien vale la pena un diagrama útil hecho a mano antes que uno inútil en 3D.

- Una clase que no tiene comportamiento no está comunicando qué rol cumple en la solución: o está faltando definir qué le puedo pedir o esa clase no debería estar en el diagrama.

- Cuando no estemos seguro de los parámetros que vamos a necesitar, a veces es mejor concentrarnos en definir el nombre del método a secas:



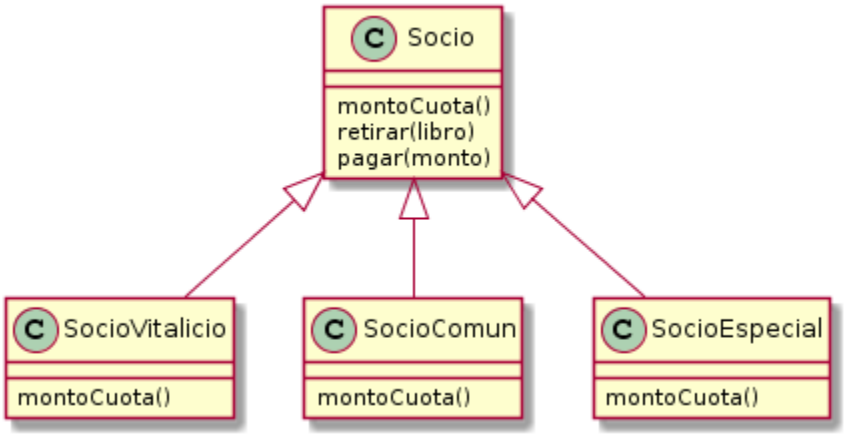
No obstante, no debemos olvidarnos de:

- Codificar el método pagar para entender qué parámetros esperamos recibir, o bien
- Volver sobre el diagrama de clases antes de considerar terminado el diseño y tomarnos un tiempo para pensar qué información va a necesitar la infracción de tránsito para resolver esa responsabilidad.

Ojo, no debe entenderse la frase anterior como: “los parámetros de un método no hay que definirlos nunca”, sino a no obligarse a definir antes de tiempo todo.

2.3 Relaciones entre clases

2.3.1 RELACIÓN “HEREDA”

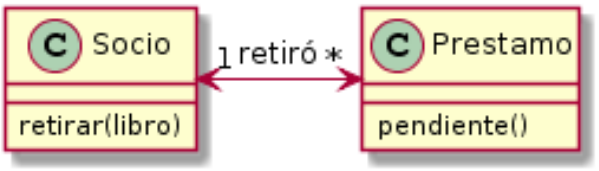


También llamada de **Generalización**: una clase específica hereda los *atributos, relaciones, operaciones y métodos* de otra más general. Es una relación que se da entre clases, no entre objetos.

Cuando una subclase redefine el comportamiento de su superclase, se escriben los nombres de los métodos que redefine (el SocioVitalicio, por ejemplo, redefine el método montoCuota()).

La flecha termina en un triángulo cerrado, es importante diferenciarla de las relaciones de asociación porque lo que comunican son cosas totalmente diferentes.

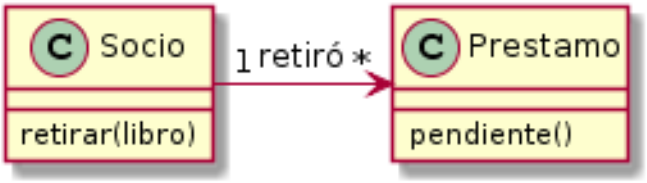
2.3.2 RELACIÓN “CONOCE”



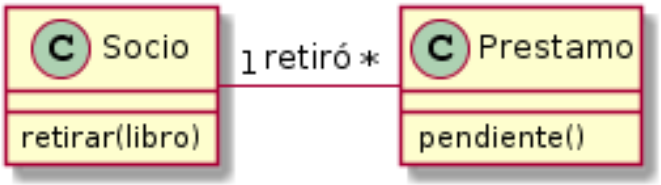
También llamada de **Asociación:** uno de los elementos conoce al otro, para lo cual guarda una referencia como variable de instancia. Puede definirse una etiqueta que expresa el rol que cumple dicha relación. En cada extremo de la asociación puede agregarse la siguiente información:

- un nombre del rol (opcional)
- flechas de navegación: determina el conocimiento (navegabilidad) desde un objeto hacia el otro. En el ejemplo anterior, el socio conoce sus préstamos y el préstamo conoce al socio que retiró el libro.

En cambio, en el ejemplo de abajo el socio conoce al préstamo pero el préstamo no conoce al socio (no tiene una variable socio para referenciarlo):



En caso de duda pueden omitirse las flechas de navegación:



- 0, 1 ó * marca la **multiplicidad**: cuántos objetos de una clase se relacionan con la otra. En nuestro caso cada socio tiene varios préstamos (indicado por el intervalo * sin cota inferior ni superior) y un préstamo tiene un socio asociado (indicado por el entero 1). La multiplicidad se puede indicar con un rango (0..1, 2..5), un rango sin cota (0..*, 1..*), un valor (1) o una serie de valores (1, 3, 5).

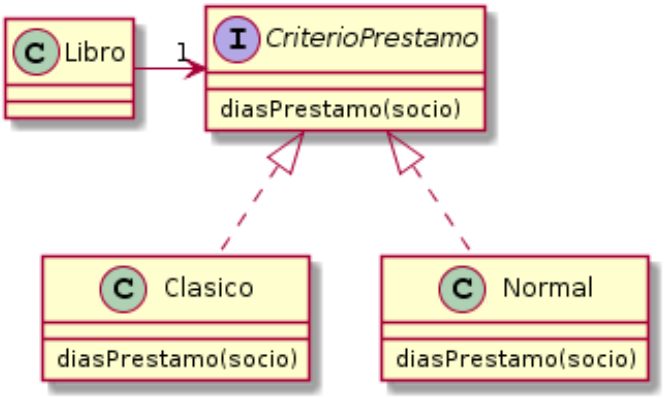
En las asociaciones, hay una relación fuerte entre ambos elementos.

2.3.3 RELACIÓN “IMPLEMENTA”

Un libro puede prestarse

- 2 días si es un clásico
- o bien 5 días - la cantidad de cuotas que adeude un socio

El criterio por el cual se presta un libro tiene un comportamiento que no requiere de una superclase. En ese caso, UML propone definir un concepto que se llama **interfaz**, que no se implementa explícitamente en Wollok, pero que muestra cuáles son las operaciones que varias clases implementan a partir de métodos:

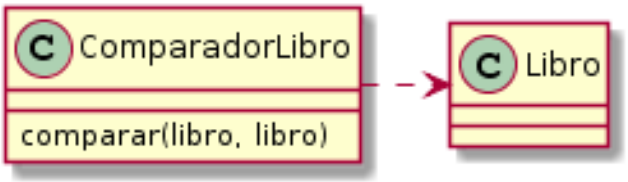


La relación es de **Realización**: se establece entre una clase y una interfaz; esto implica que la clase debe implementar todas las operaciones que defina la interfaz.

En Wollok, el libro tiene una referencia sin aclarar explícitamente las clases que lo implementan.

```
class Libro {
     var criterioPrestamo
```

2.3.4 RELACIÓN “USA”



También llamada de **Dependencia**: uno de los elementos usa o depende del otro cuando:

- El ComparadorLibro recibe o devuelve como parámetro de alguno de sus métodos un libro o
- El ComparadorLibro instancia un objeto Libro (pero no lo almacena como variable de instancia, sólo vive como variable local en el contexto de un método).

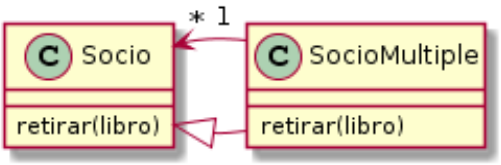
Este tipo de relación indica que los dos elementos colaboran entre sí, pero que esa relación es débil, casual; tiene un contexto temporal que no trasciende más allá de una operación. No obstante, sabemos que cualquier cambio en Libro puede impactar en alguna medida en la clase ComparadorLibro.

Esta relación se puede dar:

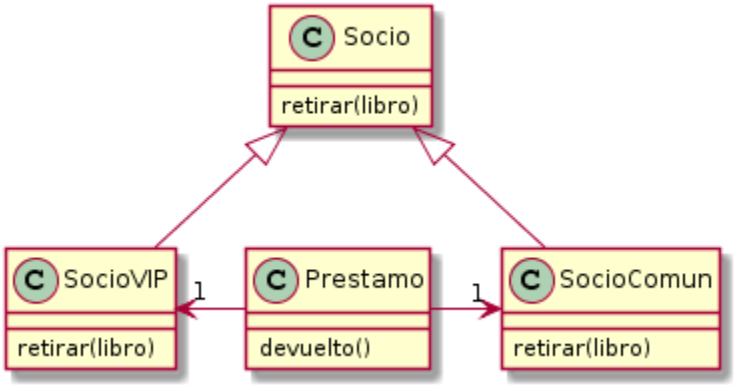
- de una clase hacia otra clase (como en el ejemplo de arriba)
- de una clase hacia una interfaz

2.4 Tips para documentar un diagrama de clases

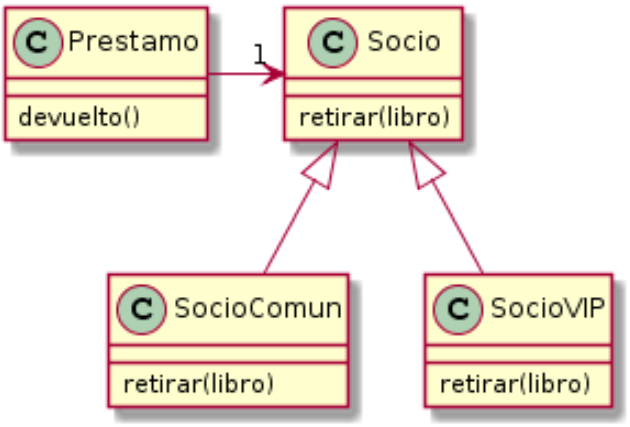
- ¿Qué pasa cuando detectamos que hay una relación de asociación y de dependencia entre dos clases? En ese caso gana la relación más fuerte: solo mostramos la relación de asociación.
- ¿Y si tenemos relaciones de asociación y de generalización? Son dos relaciones diferentes, hay que comunicar ambas.



- Al documentar una relación de asociación o dependencia, la flecha debe apuntar hacia la superclase o interfaz:

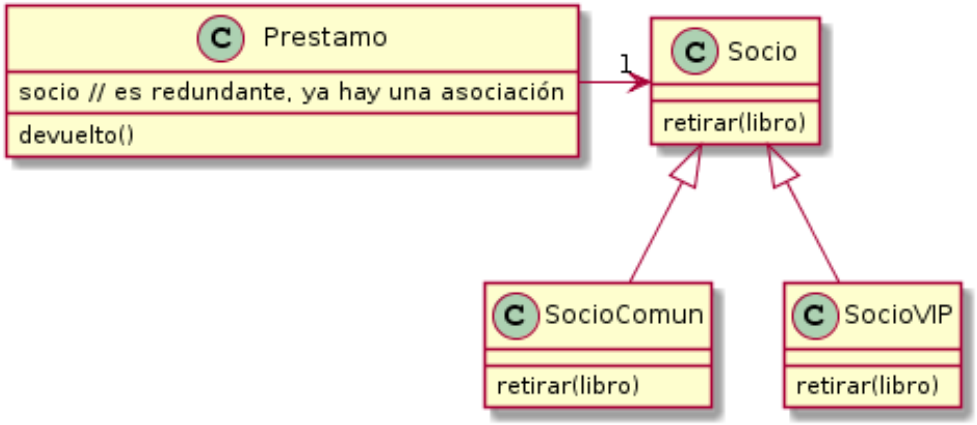


Incorrecto: Préstamo no tiene dos referencias a cada socio, sino una única referencia llamada *socio*



Correcto: la referencia de préstamo puede ser a un socio común o VIP

- En un diagrama de clases no puede haber clases que no estén relacionadas con otras
- Una flecha de asociación está marcando una referencia de un objeto a otra, es redundante escribir el atributo:

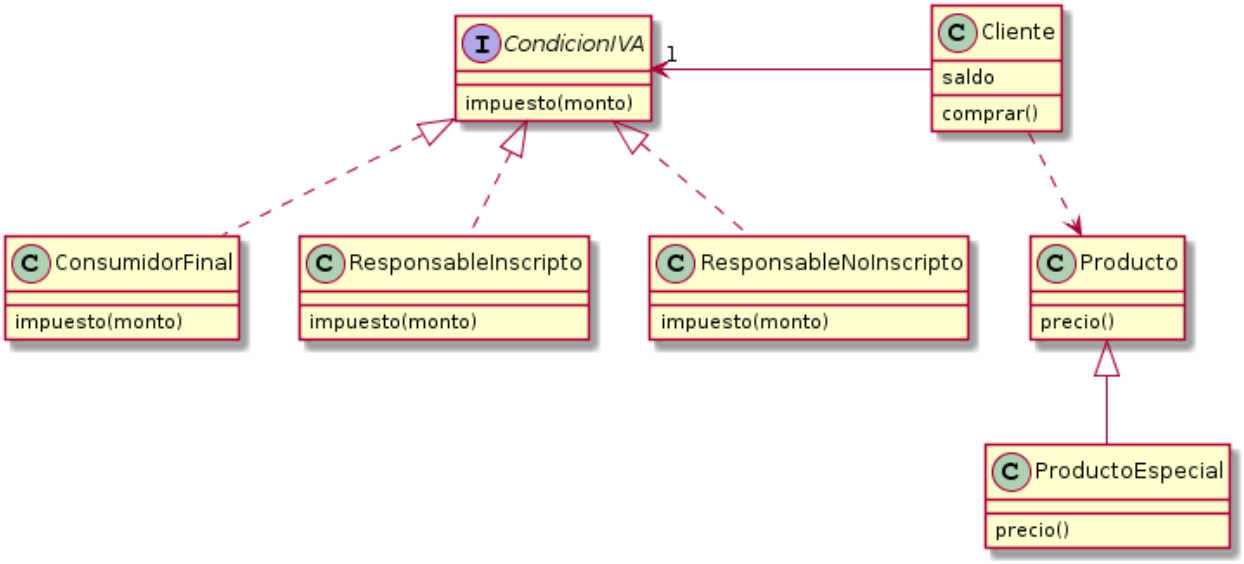


2.5 Un breve ejemplo en Wollok

```
class ConsumidorFinal  {
      method impuesto(monto) {
            ...
      }
}

class ResponsableInscripto {
      method impuesto(monto) {
            ...
```

```
	}
}

class ResponsableNoInscripto {
	method impuesto(monto) {
		...
	}
}

class Cliente {
	var condicionIVA = new ConsumidorFinal()
	var saldo

	method comprar(producto) {
		saldo += condicionIVA.impuesto(producto.precio())
		...
	}
}

class Producto {
	method precio() { ... }
}

class ProductoEspecial inherits Cliente {
	override method precio() { ... }
}
```

Nos queda el siguiente diagrama UML:



3 Diagrama de Objetos

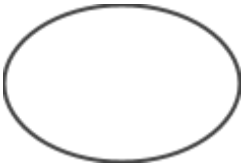
Mientras que el diagrama de clases sirve para mostrar el diseño general de una solución, el diagrama de objetos es un *snapshot* o foto del sistema en un momento determinado.

En la materia no solemos utilizar el diagrama que propone UML para armar esos casos, sino otra versión que se ha convertido en un estándar de facto por los docentes de varias universidades.

3.1 Objeto

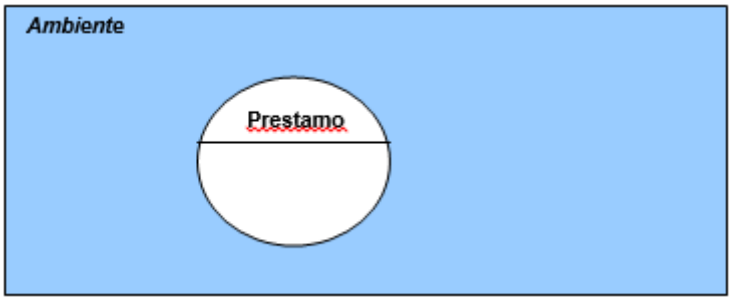
Se representa con un círculo. Si el objeto pertenece a una clase se divide en dos partes: arriba se anota el nombre de la clase a la que pertenece y abajo van las referencias a los objetos que él conoce.

Si el objeto es anónimo o wko, se representa con un círculo a secas:



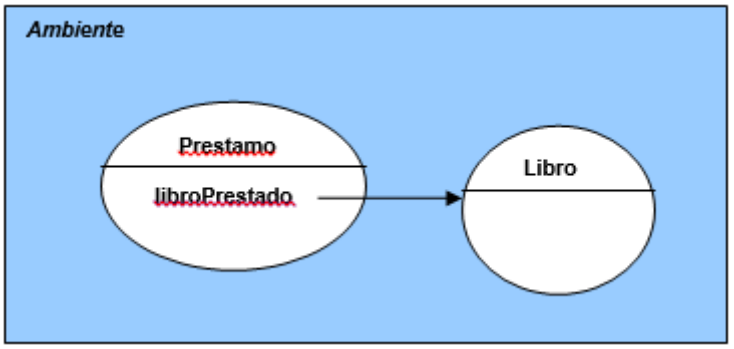
3.2 Ambiente

Es el lugar donde viven los objetos, se delimita con un rectángulo/cuadrado.

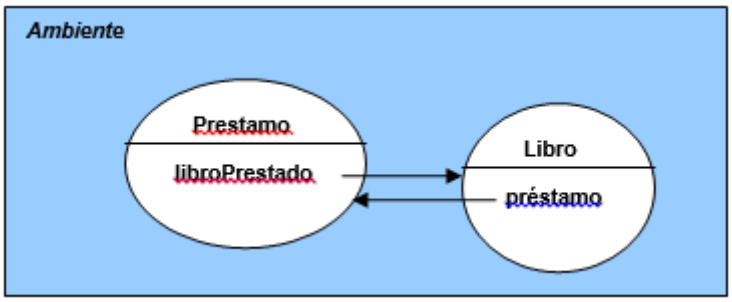


3.3 Referencias

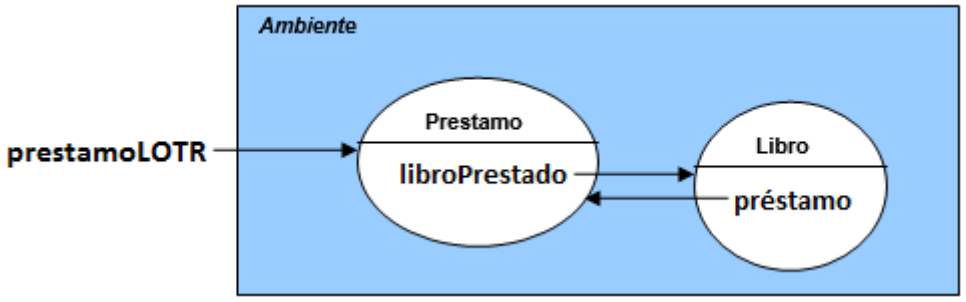
Si el *préstamo* conoce al *libro* a través de una variable de instancia *libroPrestado* esto se representa de la siguiente manera:



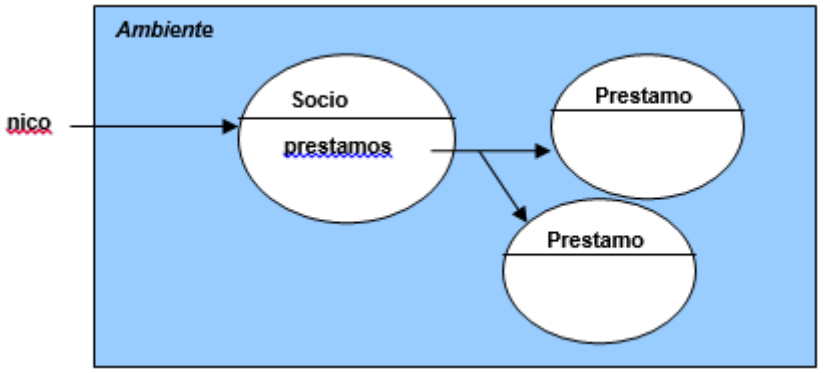
Si el libro también conoce al préstamo, lo representamos con una nueva flecha y una referencia (o variable) en Libro:



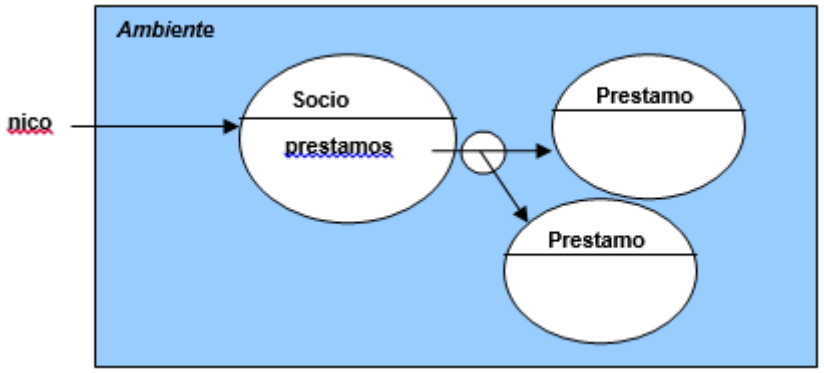
También podemos tener variables desde un workspace o test; en ese caso tenemos referencias que salen desde afuera del ambiente:



Las colecciones se pueden representar como un conjunto de flechas que salen del mismo lugar



También podemos marcar el objeto que representa la colección:



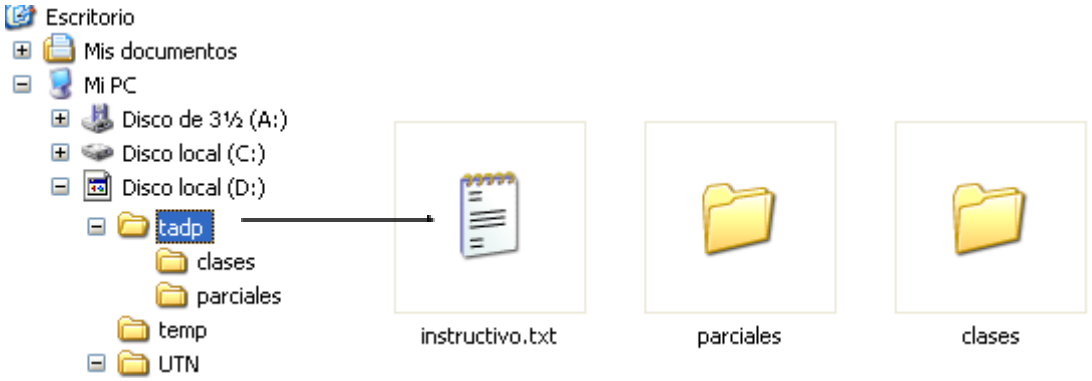
3.4 Tips para documentar un diagrama de objetos

- No olvidarse de darle nombres representativos a las variables (es mejor buenosAires que ciudad23, y es preferible eberLudueña antes que jugador_2).
- Indicar siempre a qué clase pertenece cada objeto.
- El grafo en un diagrama de objetos es dirigido, por lo tanto, las flechas deben indicar qué objeto referencia al otro.
- Si la solución involucra varios escenarios posibles, elegir los casos más representativos para modelar con el diagrama de objetos.

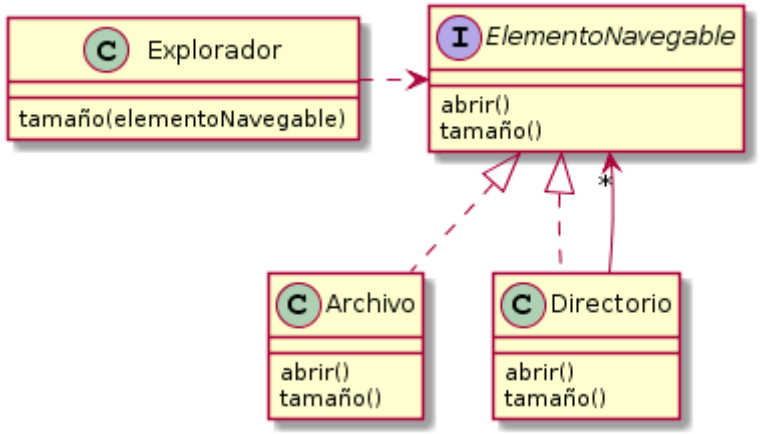
- De la misma manera que una clase no puede quedar inconexa en un diagrama de clases, tampoco puede quedar un objeto sin ser referenciado por alguien en el diagrama de objetos.
- Vale la pena invertir algunas líneas en castellano para comentar el diagrama de objetos en caso de que pueda no quedar claro por sí solo.

4 Diagrama de Secuencia (BONUS)

La idea central del diagrama de secuencia es mostrar cómo interactúan los objetos en el tiempo. Veremos sus elementos a través de un ejemplo: estamos en el explorador de Windows y queremos saber el tamaño que ocupa un directorio:



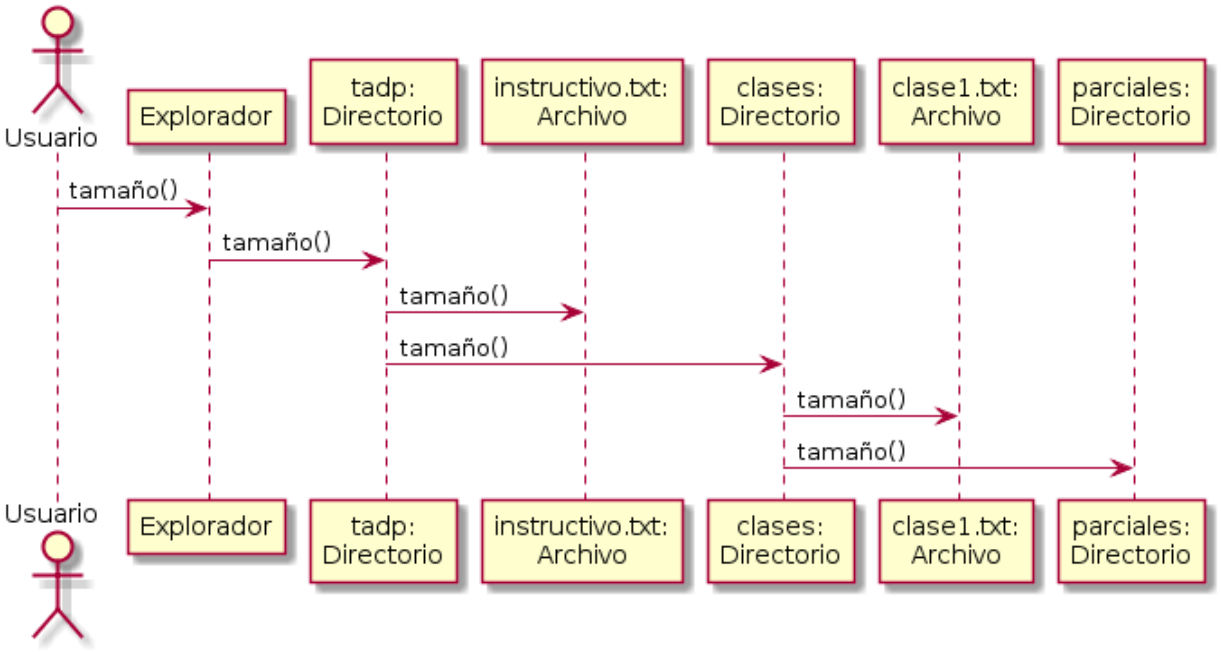
El tamaño de un directorio se define en base al tamaño de los archivos de ese directorio + el tamaño de cada subdirectorios que tiene…



Podemos hacer un diagrama de secuencia con el ejemplo de arriba, suponiendo que queremos saber el tamaño del directorio tadp, que tiene esta estructura:

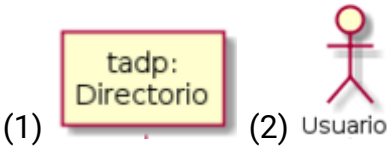
- tadp
  - clases (directorio)
    - clase1.txt (archivo)
  - parciales (directorio)
  - instructivo.txt (archivo)

Quien inicia el “caso de uso” o el requerimiento es el usuario que maneja el Explorador, enviando el mensaje tamaño(). La secuencia de envío de mensajes es la siguiente:



A continuación explicaremos los elementos que forman parte de este diagrama:

4.1 Actores/Objetos



Arriba (y opcionalmente abajo) se ubican (1) los objetos que forman parte del sistema y (2) los usuarios (elementos externos al sistema que pueden disparar mensajes). La sintaxis para especificar un objeto es:

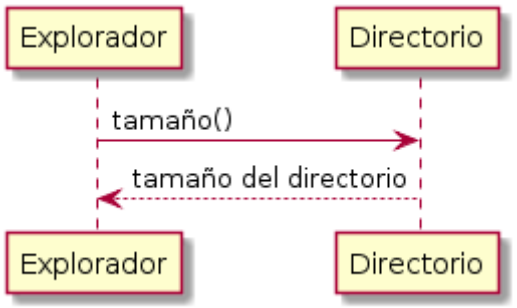
```
Nombre del objeto (opcional) : Tipo al que pertenece
```

Es conveniente poner un nombre que identifique al objeto, queda más claro de qué objeto estamos hablando, qué representa cada uno.

De la misma manera que un diagrama de objetos es una foto en un momento dado, **el diagrama de secuencia es una película de nuestro sistema planteado un escenario posible**. Entonces quienes aparecen aquí son instancias concretas que envían y reciben mensajes concretos. El diagrama nos sirve para saber quiénes preguntan, quiénes responden y cómo están repartidas las responsabilidades entre esos objetos.

4.2 Comunicación

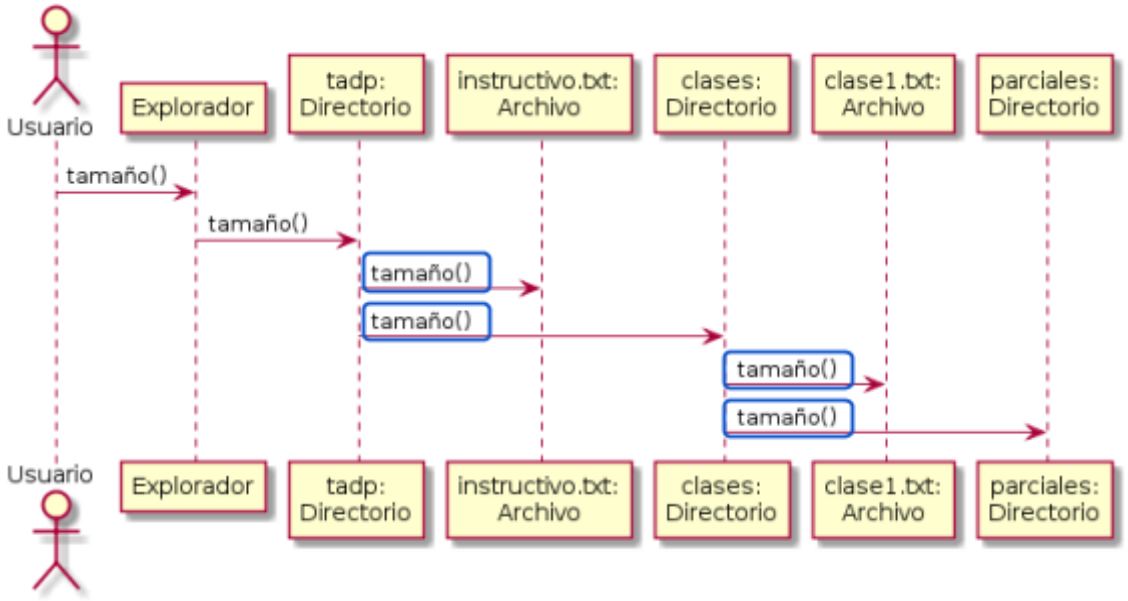
Los mensajes que se envían desde un objeto a otro se muestran con una línea que va desde el objeto emisor al receptor. Se puede mostrar lo que devuelve cada mensaje invirtiendo el sentido de la comunicación



Si bien el diagrama contempla operaciones repetitivas, en general los mensajes se envían a objetos diferentes. Por ejemplo, para calcular el tamaño de un directorio el código sería algo como:

```
method tamaño() =
    elementosNavegables.sum { elemento => elemento.tamaño() }
```

En el diagrama de secuencia, no obstante, lo que se muestra es el envío de mensajes polimórficos a distintos objetos:



Aquí vemos que la solución se describe:

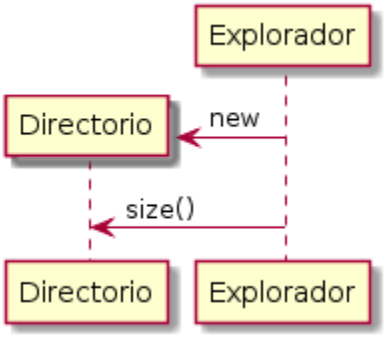
- **estáticamente**: a través del código

- **dinámicamente**: mostrando ejemplos concretos a través del diagrama de secuencia

La naturaleza del diagrama de secuencia es mostrar interacción con ejemplos concretos: tratar de definir objetos genéricos y simular el comportamiento de código es una práctica desaconsejable (para ello es mucho mejor utilizar código).

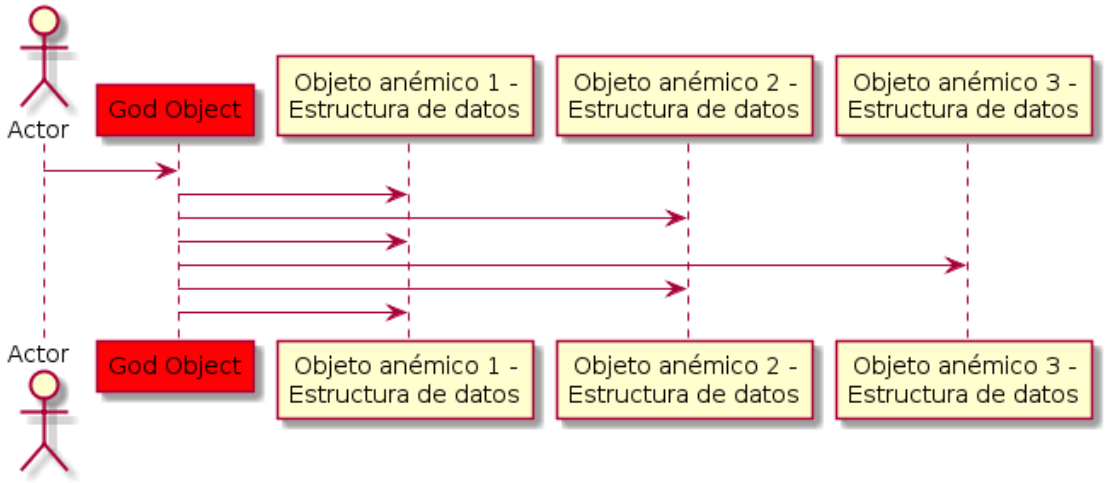
4.3 Creación de objetos

Cuando un objeto crea otro a través de un mensaje se muestra de la siguiente manera



4.4 Tips para documentar un diagrama de secuencia

- Para poder enviar un mensaje a un objeto el objeto que pregunta debe conocerlo. Esto parece obvio, pero debe ser consistente con el diagrama de clases.
- Conviene definir un diagrama de secuencia para los escenarios más representativos de un caso de uso (no necesariamente para todos)
- Un diagrama de secuencia permite descubrir objetos que preguntan mucho y objetos que no tienen responsabilidad. Lo mostramos gráficamente:



Por un lado, hay un objeto que está preguntando demasiadas cosas a los demás y en consecuencia posiblemente sea un objeto que tenga demasiada responsabilidad (lo que en la jerga de diseño se conoce como God Object). Esto va en detrimento de los objetos que reciben los mensajes: los objetos anémicos de comportamiento, simples estructuras de datos (sólo puedo consultar sus propiedades mediante getters pero no tienen lógica de negocio).

El inconveniente de este tipo de soluciones es que cualquier cambio en un objeto afecta a una gran cantidad de otros que lo conocen demasiado.

5 Resumen

En el presente documento hemos presentado diferentes formas de comunicar el diseño de una solución: además del código, que es una excelente forma de bajar a tierra ciertas decisiones, contamos con UML que es un lenguaje pensado para especificar diseños, que presenta

- diagramas estáticos, como el diagrama de clases, que exhibe la estructura y las características más estables de un sistema
- diagramas dinámicos, como el diagrama de objetos, que permite contar a partir de ejemplos cómo es la relación entre los componentes y el diagrama de secuencia, que muestra la interacción de dichos componentes a lo largo del tiempo.

Estudiar UML no nos garantiza un buen diseño, pero aplicarlo correctamente permite que nuestra solución pueda ser entendida y validada rápidamente por cualquier interlocutor, no importa el rol que ocupe dentro del equipo de desarrollo.

6 Bibliografía complementaria

Learning UML, Sinan Si Alhir, Ed. O'Reilly, julio 2003.