1 Orientación a objetos

En la vida real, podemos decir que todo el mundo sabe lo que es un objeto: algo tangible que podemos sentir y manipular. Los primeros objetos con los que interactuamos son típicamente juguetes de bebé. Los bebés aprenden rápidamente que ciertos objetos pueden hacer ciertas cosas: Los triángulos caben en hoyos en forma de triángulo, las campanas suenan o los botones se presionan.

La definición de un objeto en desarrollo de software no es muy diferente. Los objetos no son típicamente cosas tangibles que podemos levantar o sentir, pero son modelos de algo que puede hacer ciertas cosas o a los cuales se les pueden hacer ciertas cosas. **Formalmente, un objeto es una colección de datos y comportamientos asociados**.

Teniendo presente la definición de objeto, ¿qué significa entonces “orientado a objetos”? Orientado a objetos significa “funcionalidad dirigida a modelar objetos”. Es una de muchas técnicas que se usan para modelar sistemas complejos, describiendo una colección de objetos que interactúan por medio de sus datos y comportamiento.

Cuando leemos sobre orientación a objetos, es probable que nos crucemos con los términos “análisis orientado a objetos”, “diseño orientado a objetos” y “programación orientada a objetos”. Todos estos conceptos están relacionados bajo la idea general de orientación a objetos. De hecho, el análisis, el diseño y la programación son etapas del proceso de desarrollo de software; ponerles el apellido de “orientación a objetos” simplemente especifica el estilo de desarrollo de software que se busca implementar.

El **análisis orientado a objetos** es el proceso de analizar un problema, sistema o tarea que alguien quiere resolver o convertir en una aplicación, identificando los objetos y las interacciones entre ellos. La etapa de análisis se refiere a **qué** se necesita hacer. El resultado de la etapa de análisis es un conjunto de requisitos.

El **diseño orientado a objetos** es el proceso de convertir los requisitos en una especificación de implementación. El diseñador debe nombrar los objetos, definir sus comportamientos y especificar formalmente las interacciones entre los objetos. Esta etapa se refiere a **cómo** se deben hacer las cosas. El resultado de la etapa de diseño es una especificación de la implementación donde los requisitos se convierten en clases e interfaces que pueden ser implementadas en (idealmente) cualquier lenguaje de programación orientada a objetos.

La **programación orientada a objetos** es el proceso de convertir el diseño definido en un programa funcional que hace exactamente lo que se requería en principio.

Sería genial si el mundo fuera ideal y pudiéramos seguir estas etapas una a una, en perfecto orden. Sin embargo, el mundo real es más complejo y, sin importar que tanto intentemos separar estas etapas, siempre encontraremos cosas que necesitan análisis adicional mientras estamos diseñando. Cuando estamos programando, encontramos características que necesitan más diseño. En el mundo actual, la mayoría del desarrollo ocurre siguiente un modelo iterativo e incremental.

En el desarrollo iterativo e incremental, una pequeña parte del sistema es modelada, diseñada y programada, luego el programa es revisado y expandido para mejorar cada funcionalidad e incluir nuevas características en una serie de ciclos cortos.

Aunque este curso está enfocado en programación orientada a objetos, no podemos dejar de lado el análisis y el diseño, ya que son actividades necesarias para que los programas que escribimos solucionen adecuadamente los problemas y necesidades que inicialmente se plantean.

En el resto de este documento abordaremos los conceptos básicos de orientación a objetos que nos permitirán entender de forma general el funcionamiento de la orientación a objetos sin necesidad de lidiar con sintaxis de software o intérpretes de código. Estos aspectos más técnicos los iremos abordando más adelante en el curso, a medida que nos adentremos más en el mundo de la programación orientada a objetos.

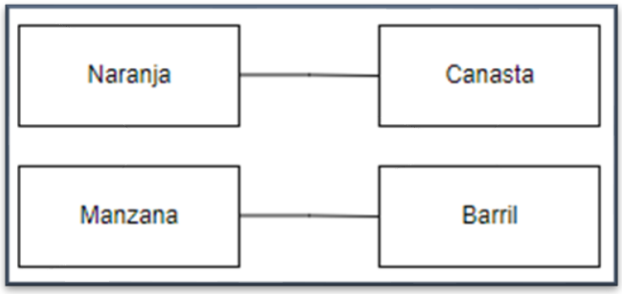
2 Objetos y clases

Un objeto es una colección de datos y comportamientos asociados. ¿Cómo distinguimos un objeto de otro? Vamos a suponer que estamos haciendo un sistema de inventario para una granja de frutas que vende manzanas y naranjas. Podemos asumir que las manzanas se guardan en barriles y las naranjas en canastas.

En este ejemplo, tenemos cuatro tipos de objetos: manzanas, naranjas, barriles y canastas. En modelado orientado a objetos, el término usado para tipos de objetos es **clase**. En términos técnicos, ahora tenemos cuatro **clases** de objetos.

¿Cuál es la diferencia entre un objeto y una clase? Las clases describen objetos. Las clases son como los planos para crear un objeto. Podemos tener tres naranjas en la mesa; cada naranja es un objeto diferente, pero todas tres tienen los atributos y el comportamiento asociado con una clase: la clase general de naranjas.

Las relaciones entre las cuatro clases de objetos en nuestro sistema de inventario pueden describirse usando un **diagrama de clases** del **Lenguaje de Modelado Unificado** (UML por sus siglas en inglés). Este sería nuestro diagrama:



Este diagrama simplemente muestra que una naranja está asociada de alguna forma con una canasta y que una manzana está asociada de alguna forma con un barril. Una **asociación** es la forma más básica de relacionar dos clases.

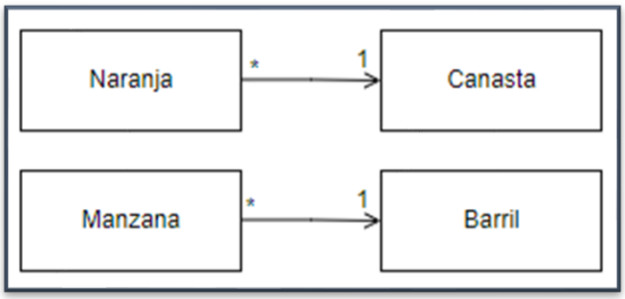
UML es muy popular entre directores y administradores, pero es ocasionalmente apreciado por programadores. La sintaxis de UML es generalmente muy obvia, fácil de dibujar y muy intuitiva. Tener un estándar basado en estos diagramas intuitivos hace más fácil que los programadores se puedan comunicar entre ellos, los diseñadores y administradores.

Sin embargo, algunos programadores piensan que UML es una pérdida de tiempo, ya que argumentan que las especificaciones formales hechas en diagramas UML serán redundantes e innecesarias antes de ser implementadas. Además, mantener esos diagramas solo representa una pérdida de tiempo y no trae ningún beneficio.

Esto es cierto para algunas organizaciones y no tan cierto para otras. Sin embargo, cada equipo de desarrollo conformado por más de una persona se tendrá que sentar ocasionalmente para discutir los detalles de partes del sistema en el que están trabajando. UML es extremadamente útil en estas sesiones para comunicar las ideas fácil y rápido.

Por otra parte, la persona más importante con la que tendremos que comunicarnos es con nosotros mismos. Nosotros pensamos que podemos recordar las decisiones de diseño que hemos tomado, pero siempre tendremos en nuestro futuro esos momentos en los que nos preguntamos "¿por qué hice esto?". Si mantenemos los diagramas iniciales donde bosquejamos nuestros diseños, eventualmente nos daremos cuenta de que son referencias muy útiles.

Volviendo a nuestro diagrama inicial, aunque es correcto, no nos indica adecuadamente que las manzanas van en barriles o en cuántos barriles puede ir una sola manzana. La asociación entre clases es frecuentemente obvia y no requiere mayor explicación, pero siempre tenemos la opción de agregar información adicional para clarificarla. En UML solo necesitamos especificar tanta información en un diagrama como consideremos necesaria para que tenga sentido en la situación actual. En el caso de las manzanas y los barriles, podemos estar seguros de que la asociación es "muchas manzanas van en un barril", pero para asegurarnos de que nadie se confunda, podemos mejorar el diagrama así:



Este diagrama nos indica que las naranjas van en canastas con una pequeña flecha. También nos indica la **multiplicidad** (número de objetos que se pueden usar en la asociación) en ambos lados de la relación. Una canasta puede tener muchos (representado por un *) objetos de tipo naranja. Una naranja puede ir en exactamente una canasta.

Para la relación entre manzana y barril, si la leemos de izquierda a derecha, muchas instancias de la clase manzana (muchos objetos de tipo manzana) pueden ir en un solo barril. Si la leemos de derecha a izquierda, exactamente un barril puede estar asociado con una manzana.

3 Especificando atributos y comportamientos

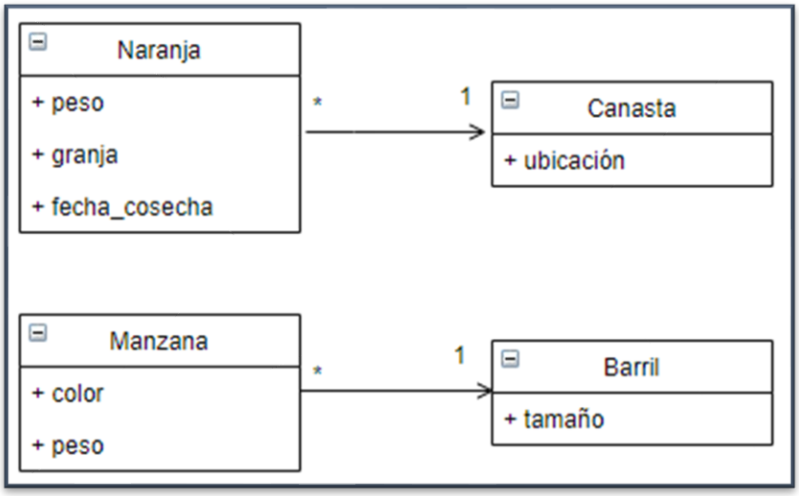
Ya tenemos una comprensión breve de algunos términos de orientación a objetos. Los objetos son **instancias** de clases que pueden estar asociadas entre sí. Una instancia es un objeto específico con su propio conjunto de datos y comportamientos. Por ejemplo, una naranja específica en la mesa en frente de nosotros es una instancia de la clase general de naranjas. Ese concepto es lo suficientemente simple, pero ¿qué son esos datos y comportamientos que están asociados con cada objeto?

3.1 Los datos describen los objetos

Los datos representan las características individuales de cierto objeto. Una clase de objetos puede definir características específicas que son compartidas por todas las instancias de esa clase. Cada instancia puede tener diferentes valores para las características dadas. Por ejemplo, nuestras tres naranjas en la mesa podrían tener un peso diferente cada una. La clase naranja podría tener entonces un **atributo** peso. Todas las instancias de la clase naranja tienen un atributo peso, pero cada naranja puede tener diferentes valores por ese peso. Sin embargo, los atributos no tienen que ser únicos; cualesquiera dos naranjas pueden tener el mismo peso.

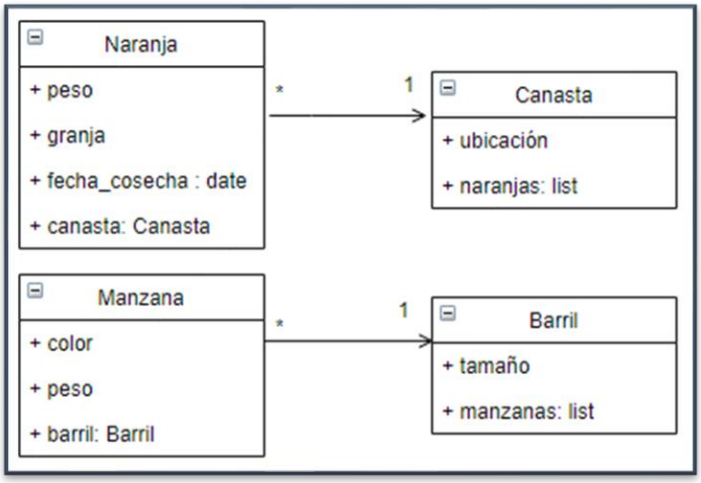
A los atributos se les refiere frecuentemente como **propiedades**. Algunos autores sugieren que los dos términos tienen significados diferentes: usualmente los atributos son modificables, mientras que las propiedades son de solo lectura. En Python, el concepto de “solo lectura” no es realmente utilizado, entonces en este curso usaremos los dos términos indistintamente. Además, la palabra clave propiedad (**property**) tiene un significado especial en Python para un tipo particular de atributo que veremos más adelante en el curso.

En nuestra aplicación de inventario de frutas, el granjero puede querer conocer la huerta de la que viene la naranja, cuándo fue cosechada y cuánto pesa. También puede querer llevar un registro de donde se almacena cada canasta. Las manzanas pueden tener un atributo color y los barriles pueden tener diferentes tamaños.



Dependiendo de qué tan detallado necesite ser nuestro diseño, también podemos especificar el tipo de cada atributo. Los tipos de atributo son frecuentemente datos primitivos que son estándares en la mayoría de los lenguajes de programación, como son entero (int), real (float), texto (string) o lógico (boolean). Sin embargo, los tipos también pueden representar estructuras de datos como listas, árboles o grafos o, incluso, otras clases.

En nuestro ejemplo, hasta el momento, los atributos son todos datos primitivos. Pero hay atributos implícitos que podemos volver explícitos: las asociaciones. Por ejemplo, para una naranja dada, podríamos tener un atributo que contenga la canasta donde está la naranja. De forma alternativa, una canasta podría tener una lista de las naranjas que contiene. El siguiente diagrama agrega estos atributos e incluye la descripción de los tipos.



3.2 Los comportamientos son acciones

Los comportamientos son las acciones que pueden ocurrir en un objeto. Los comportamientos que se pueden llevar a cabo en una clase específica de objetos son llamados **métodos**. Al nivel de la programación, los métodos son como funciones en la programación estructurada, pero tienen acceso a todos los datos asociados con el objeto al que pertenecen. Al igual que las funciones, lo métodos pueden aceptar **parámetros** y **retornar** valores.

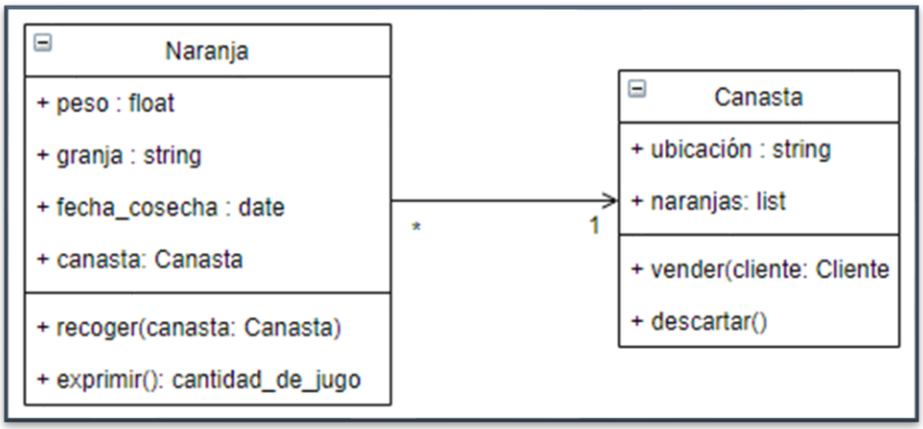
Los parámetros de un método son una lista de objetos que deben ser pasados al método que está siendo invocado. Estos objetos son usados por el método para llevar a cabo el comportamiento o tarea que debe hacer. Los valores de retorno son el resultado de dicha tarea.

Vamos a extender nuestro ejemplo del sistema de inventario de frutas. Una acción que podría estar asociada con las naranjas es la acción de **recoger**. Si pensamos en la implementación, la acción recoger pondría la naranja en la canasta actualizando el atributo canasta en la naranja y adicionando la naranja a la lista de naranjas en la canasta. Entonces recoger necesita conocer la canasta con la que está trabajando. Podemos lograr esto pasándole un parámetro canasta al método recoger.

Ahora supongamos que el granjero de frutas también vende jugo. Podemos agregar un método **exprimir** a la naranja. Cuando se exprime la naranja, el método exprimir puede retornar la cantidad de jugo obtenido y remover la naranja de la canasta en la que estaba.

Otra suposición que podemos hacer para el ejemplo es que la canasta podría tener una acción **vender**. Cuando se vende una canasta, nuestro sistema de inventario podría actualizar algunos datos en algún objeto que aún no hemos especificado para la contabilidad y cálculo de las ganancias. Por otro lado, nuestra canasta de naranjas se podría dañar antes de ser vendida, así que podemos agregar un método **descartar**.

La nueva versión del diagrama quedaría así:



Agregar datos y métodos a objetos individuales nos permite crear un sistema de objetos que interactúan entre sí. Cada objeto en el sistema es un miembro de cierta clase. Dichas clases especifican los tipos de datos que el objeto puede guardar y los métodos que pueden ser invocados en el objeto. Los datos en cada objeto pueden estar en diferentes estados de otros objetos de la misma clase y cada objeto puede reaccionar diferente a las llamadas a métodos debido a estas diferencias en estado.

El análisis y el diseño orientado a objetos consiste básicamente en descubrir cuáles son los objetos que hacen parte del sistema y cómo deben interactuar entre sí.

4 Ocultando los detalles y creando la interfaz pública

El principal propósito de modelar un objeto en el diseño orientado a objetos es determinar cuál es la **interfaz** pública de dicho objeto. La interfaz de un objeto es la colección de atributos y métodos que otros objetos pueden usar para interactuar con él. Los demás objetos no necesitan, y frecuentemente no se les permite, acceder a los detalles internos del objeto. Un ejemplo del mundo real es el televisor. Nuestra interfaz con el televisor es el control remoto. Cada botón en el control remoto representa un método que puede ser invocado en el objeto televisor. Cuando nosotros accedemos estos métodos, no sabemos o no nos interesa si el televisor está obteniendo la señal de una antena, una conexión por cable o un satélite. Tampoco nos interesa saber qué señales electrónicas se están enviando para ajustar el volumen.

Este proceso de ocultar la implementación o los detalles funcionales de un objeto se conoce como **ocultamiento de información**. También se le refiere como **encapsulamiento**, aunque este término implica algunas cosas adicionales. Como desarrolladores en Python, no tenemos o necesitamos un ocultamiento de información real, sin embargo, el concepto de interfaz pública es muy importante.

La interfaz pública debe ser cuidadosamente diseñada ya que puede resultar difícil modificarla en el futuro. Cambiar la interfaz de un objeto puede afectar otros objetos cliente que la estén invocando. Podemos cambiar la funcionalidad interna todo lo que queramos, por ejemplo, para hacerla más eficiente o para acceder datos en la nube y localmente, y los objetos cliente podrán seguir interactuando con el objeto usando su interfaz pública. Por otro lado, si cambiamos la interfaz de un objeto, modificando los nombres de sus atributos o alterando el orden de los parámetros de un método, todos los objetos cliente tendrán que ser modificados también.

En este punto podemos introducir otro concepto, muy relacionado con el encapsulamiento y el ocultamiento de información, que es muy común en la orientación a objetos: la abstracción. Puesto de manera simple, la abstracción significa tratar con el nivel de detalle que es más apropiado para una tarea dada. Es el proceso de extraer una interfaz pública de los detalles internos. Por ejemplo, un conductor de un vehículo necesita interactuar con el volante, el acelerador y los frenos. El funcionamiento del motor, la caja de cambios y el subsistema de frenos no le importa al conductor. Por otro lado, un mecánico trabaja en un nivel de abstracción diferente, afinando el motor y sangrando los frenos. Esto representa dos niveles de abstracción de un carro.

Podemos simplificar todos estos términos en una sola definición: Abstracción es el proceso de encapsular información con interfaces pública y privadas separadas. Las interfaces privadas pueden estar sujetas a ocultamiento de información.

Lo más importante que debemos sacar de todo lo anterior es que debemos construir nuestros modelos de forma que sean entendibles para otros objetos que tienen que interactuar con ellos. Esto quiere decir que debemos prestar mucha atención a los pequeños detalles.