AWAKELAB

4.3.- Contenido 3: Representar un problema de orientación de objetos mediante un diagrama de clases para su implementación en Python

Objetivo de la jornada

1. Bosqueja un diagrama de clases para representar un problema distinguiendo clases, atributos, métodos y relación entre clases
2. Implementa un programa utilizando clases, atributos, métodos y relación entre ellos a partir de un diagrama de clases para dar solución a un problema

4.3.1.- Diagramas de Clase

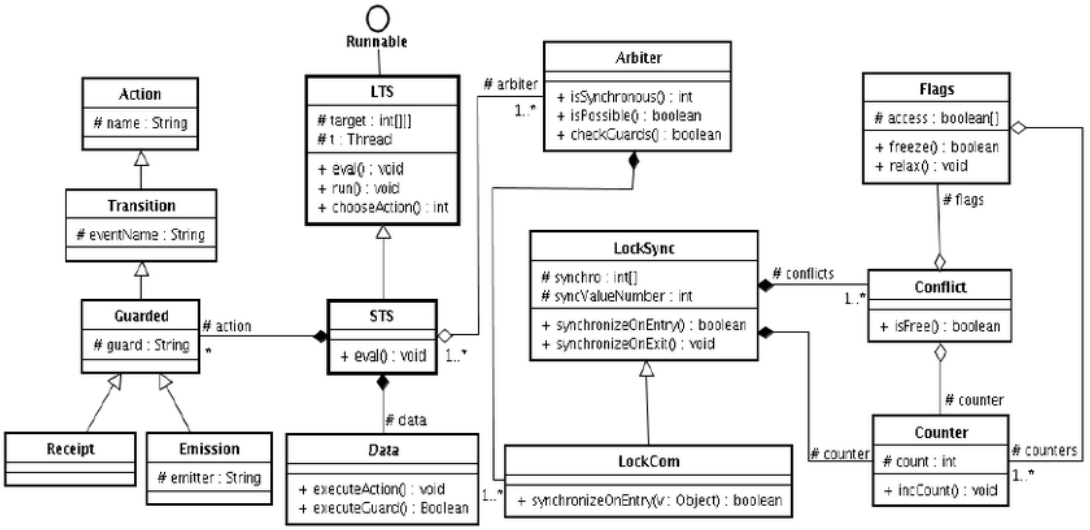
4.3.1.1. Qué es un diagrama de clases

El lenguaje de modelado unificado (UML) es el sucesor de la ola de métodos de análisis y diseño orientados a objetos (OOA&D) que apareció a finales de los 80 y principios de los 90. Unifica más directamente el método de Booch, Rumbaugh (OMT) y Jacobson, pero su alcance es más amplio. El UML pasó por un proceso de estandarización con OMG (Object Management Group) y ahora es un estándar de OMG.
Si se desvía de la notación oficial, otros desarrolladores no entenderán completamente lo que está diciendo. Sin embargo, hay ocasiones en las que la notación oficial puede interferir con sus necesidades. En estos casos, no hay que tener miedo de torcer un poco el lenguaje. El lenguaje podría torcerse para ayudar a comunicarse, y no al revés. Pero esta práctica no se hace a menudo y siempre se debe ser consciente de que una desviación es mala si causa problemas de comunicación.

4.3.1.2. Para qué sirven los diagramas de clases

Cuando se trata de programación, el verdadero objetivo del desarrollo de software es producir código. Después de todo, los diagramas son sólo imágenes bonitas. Ningún usuario le agradecerá las bonitas imágenes; lo que quiere un usuario es un software que se ejecute.
Por lo tanto, cuando esté considerando usar UML, es importante que se pregunte por qué lo está haciendo y cómo lo ayudará a la hora de escribir el código. No hay evidencia empírica adecuada que demuestre que estas técnicas sean buenas o malas, pero las siguientes subsecciones discuten las razones con las que a menudo uno se encuentra para usarlas.

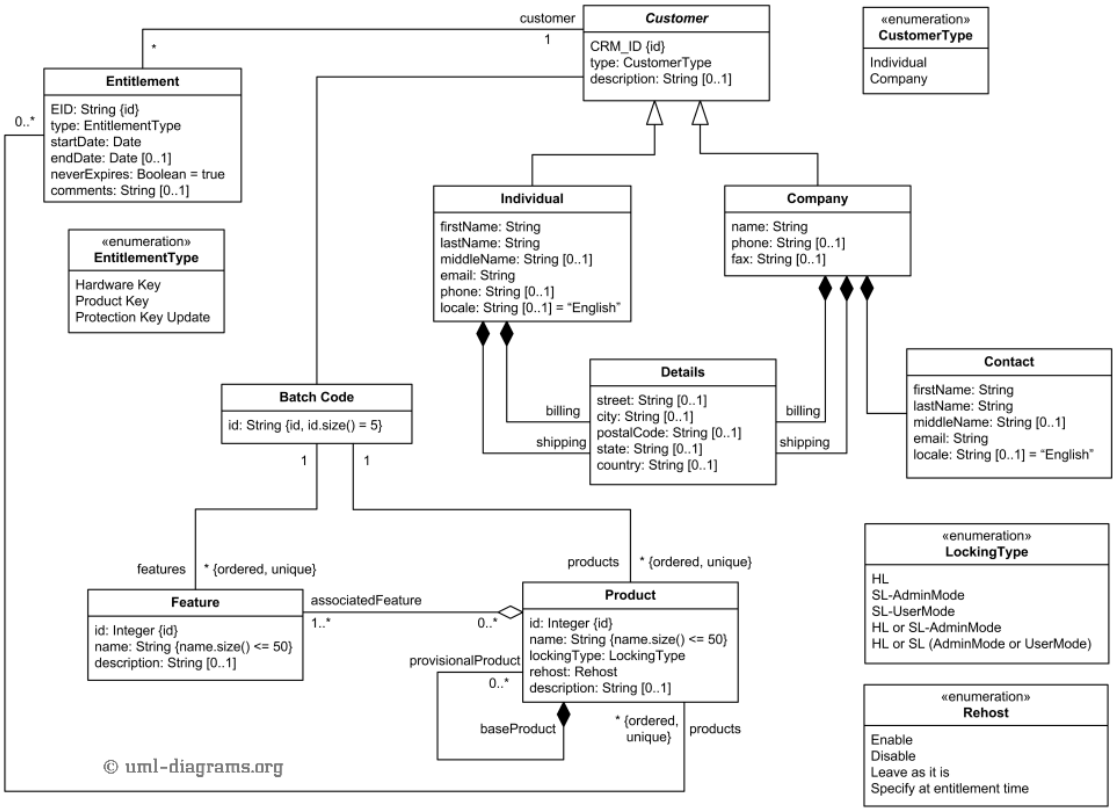
Si es la primera vez que ve un diagrama de clases UML como el de figura, probablemente encontrará la figura desconcertante, pero cuando haya terminado de leer este documento, le hará perfecto sentido (tocamos madera).



El lenguaje de modelado unificado (UML) es una herramienta útil para el análisis y el diseño orientados a objetos. Una parte del UML, el diagrama de clases, se utiliza con frecuencia porque este tipo de diagrama es especialmente bueno para describir de forma clara y sucinta las relaciones estáticas entre objetos.

4.3.1.3. Notación de un diagrama de clases

Vamos a ver la notación estandarizada usada por los diagramas.

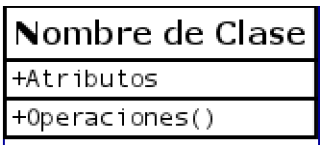


Acá vemos un ejemplo similar al de la figura anterior, solo queremos ilustrar que la notación, en la realidad, es bastante estándar, pero no es extraño encontrarse con pequeñas desviaciones del modelo.

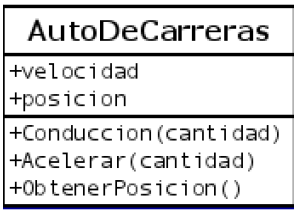
4.3.1.4. Atributos y métodos en un diagrama de clases

Nombres de clases, atributos y operaciones

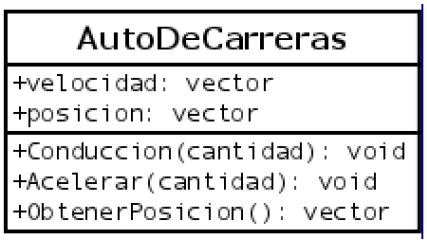
En primer lugar, comencemos con el nombre, los atributos y las operaciones de una clase. Las clases están representadas por un rectángulo dividido en tres compartimentos. El nombre de la clase está en negrita en la parte superior del rectángulo, los atributos se escriben debajo y las operaciones se enumeran en el compartimento inferior. Ver figura.



Por ejemplo, si uno de los objetos de un juego es un coche de carreras, se puede especificar como se muestra en la siguiente figura (atributos y operaciones).



Por supuesto, es probable que un objeto automóvil de carreras sea mucho más complejo que esto, pero solo necesitamos enumerar los atributos y operaciones de interés inmediato. La clase puede desarrollarse más fácilmente en una etapa posterior si es necesario. (Muy a menudo, no se muestra ningún atributo u operación en absoluto y se usan diagramas de clases simplemente para mostrar las relaciones entre objetos, como lo demuestra la primera figura) Tenga en cuenta que las operaciones de una clase definen su interfaz. El tipo de atributo se puede mostrar enumerado después de su nombre y separado por dos puntos. El valor de retorno de una operación también se puede mostrar de la misma manera, al igual que el tipo de parámetro. Ver figura:



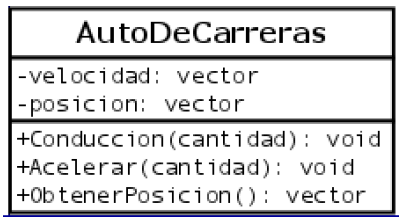
Ahora bien, el valor de retorno es requerimiento no opcional en lenguajes como C++ o Java, no obstante, sin ser un requerimiento estricto se puede también utilizar en Python, tanto para indicar variables de recepción, como variable de salida de un método.

```
from typing import List
vector = List[float]

class AutoDeCarreras():

      def __init__(self, velocidad: vector, posición: vector):
      self.velocidad = velocidad
      self.posicion = posicion

      ...

      def Conduccion(cantidad: int) -> vector

      ...
```

En este apartado, rara vez utilizamos el formato "nombre: tipo" para los parámetros, ya que a menudo hace que los diagramas sean demasiado grandes para caber cómodamente en la página. En su lugar, solo enumeramos el tipo o, a veces, un nombre descriptivo si el tipo se puede inferir de él. No obstante tenga presente, que los diagramas UML, pueden ser usados para realizar implementaciones en cualquier lenguaje, y algunos son más rigurosos que otros en este sentido.

Visibilidad de atributos y operaciones

Cada atributo de clase y operación tiene una visibilidad asociada. Por ejemplo, un atributo puede ser público, privado o protegido. Esta propiedad se muestra usando los símbolos '+' para público, '-' para privado y '#' para protegerse'. La Figura muestra el objeto AutoDeCarreras con la visibilidad de sus atributos y operaciones enumeradas.



Note que las variables 'velocidad' y 'posición' son privadas (recuerde que como ya vimos Python tiene convenciones con guiones bajos para tales propósitos)
Al igual que con los tipos, no es imperativo que enumere las visibilidades al dibujar diagramas de clases; sólo necesitan mostrarse si son de importancia inmediata para la parte del diseño que está modelando (o, como en nuestro caso, describiendo).

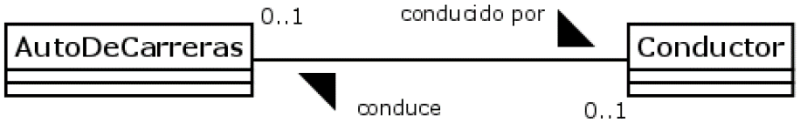
```
...

class AutoDeCarreras:

      def __init__(self, velocidad: int, posicion: int):
          self.__velocidad = velocidad
          self.__posicion = posicion

      def Conduccion(self, cantidad: int) -> None:
          parametro_conduccion = cantidad
          '''codigo'''

      def Acelerar(self, cantidad: int) -> None:
          parametro_acel = cantidad
          '''codigo'''

      def ObtenerPosicion(self) -> dict:
          '''Codigo de Calculos'''
          return result

      ...
```

Relaciones

Las clases no sirven de mucho por sí solas. En el diseño orientado a objetos, cada objeto generalmente tiene una relación con uno o más objetos, como la relación de herencia de tipo hijo-padre o la relación entre un método de clase y sus parámetros. A continuación, se describe la notación que UML específica para indicar cada tipo particular de relación.

Asociación

Una asociación entre dos clases representa una conexión o vínculo entre instancias de esas clases y se indica mediante una línea continua. Desafortunadamente, en el momento de escribir este apartado, los practicantes de UML parecen incapaces de ponerse de acuerdo sobre lo que realmente significa el texto de la oración anterior, por lo que para los propósitos de esta sección se dice que existe una asociación entre dos clases si una de ellas contiene una referencia persistente a la otra.
La Figura siguiente muestra la asociación entre un objeto AutoDeCarreras y un objeto Conductor.

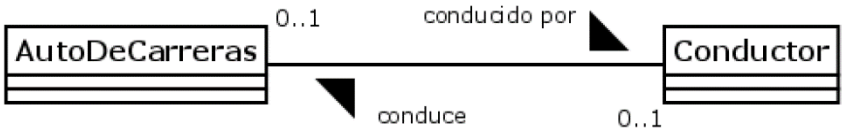


Este diagrama de clases nos dice que un auto de carreras es conducido por un conductor y que un conductor conduce un auto de carreras. También nos dice que

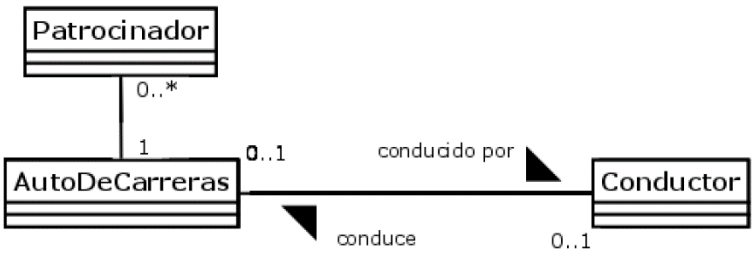
una instancia de AutoDeCarreras mantiene una referencia persistente a una instancia de Conductor (a través de un puntero, instancia o referencia) y viceversa. En este ejemplo, ambos extremos han sido nombrados explícitamente con una etiqueta descriptiva llamada nombre de rol, aunque la mayoría de las veces esto no es necesario ya que el rol suele estar implícito dado los nombres de las clases y el tipo de asociación que las vincula. Se prefiere nombrar los roles solo cuando se cree que es absolutamente necesario, ya que, hace que un diagrama de clases complejo sea más simple de comprender.

Multiplicidad

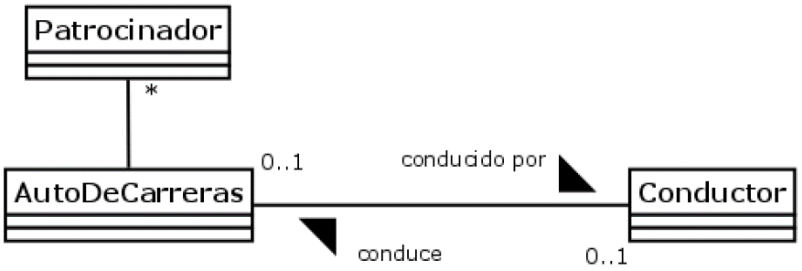
El final de una asociación también puede tener multiplicidad, lo que es una indicación del número de instancias que participan en la relación. Por ejemplo, un automóvil de carreras solo puede tener uno o ningún conductor, y un conductor conduce un automóvil o no. Esto se puede mostrar como en la Figura usando 0..1 para especificar el rango.



La figura siguiente demuestra cómo se puede mostrar que un objeto AutoDeCarreras está asociado con cualquier número de objetos Patrocinador (usando un asterisco para indicar infinito como el límite superior del rango), y cómo un Patrocinador solo puede asociarse con un AutoDeCarreras en cualquier momento.



La Figura anterior muestra la forma general de especificar un rango ilimitado y un rango de 1, pero a menudo verá estas relaciones expresadas en forma abreviada, como se muestra en la Figura de abajo. El asterisco único denota un rango ilimitado entre 0 e infinito, y la ausencia de números o un asterisco al final de una asociación implica una relación singular.

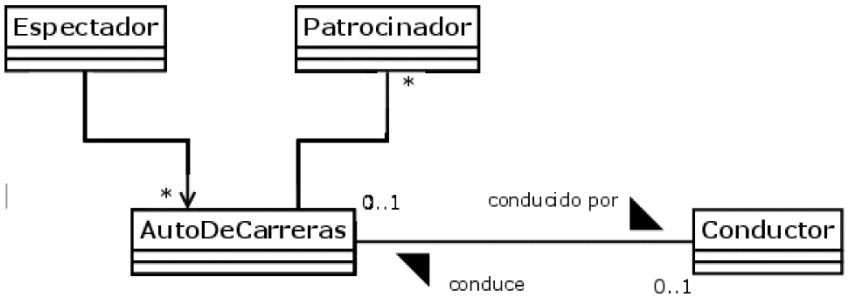


También es posible que una multiplicidad represente una combinación de valores discretos. Por ejemplo, un automóvil puede tener dos o cuatro puertas: 2, 4. Dadas solo las asociaciones que se muestran en la Figura, podemos inferir cómo una interfaz para una clase AutoDeCarreras podría verse:

```
class AutoDeCarreras:

  def GetConductor(self) -> Conductor:
       pass

  def SetConductor(self, conductor: Conductor)-> None:
       pass

  def EstaSiendoConducido(self)-> bool:
       pass

  def AgregarPatrocinador(self, patrocinador: Patrocinador)-> None:
       pass

  def EliminarPatrocinador(self, patrocinador: Patrocinador)-> None:
       pass

  def GetNumeroDePatrocinadores(self) -> int:

       pass

       ...
```

Navegabilidad

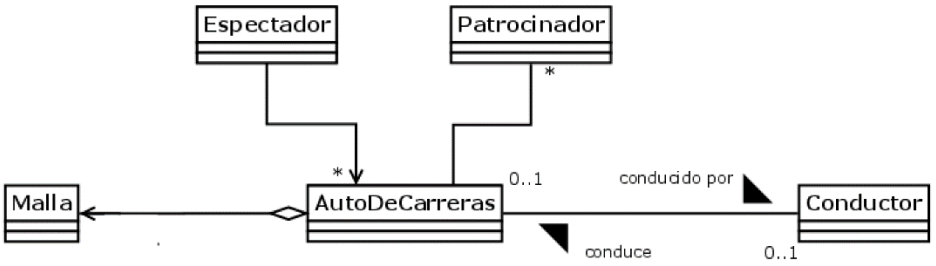
Hasta ahora, las asociaciones que hemos visto han sido bidireccionales: un AutoDeCarreras conoce una instancia de Conductor y esa instancia de Conductor conoce la instancia de AutoDeCarreras. Un AutoDeCarreras conoce cada instancia de Patrocinador y cada Patrocinador conoce el AutoDeCarreras. Sin embargo, a menudo necesitará expresar una asociación unidireccional. Por ejemplo, es poco probable que un AutoDeCarreras deba estar al tanto de los espectadores que lo miran, pero es importante que un espectador sea consciente del automóvil que está mirando. Esta es una relación unidireccional y se expresa agregando una flecha al extremo apropiado de la asociación. Ver figura



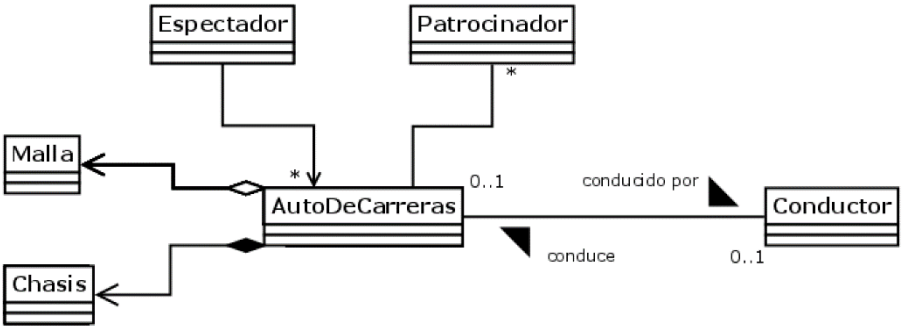
Observe también cómo la figura muestra claramente cómo un espectador puede estar mirando cualquier número de autos de carreras.

Agregación compartida y compuesta

La agregación es un caso especial de asociación y denota la parte de la relación. Por ejemplo, un brazo es parte de un cuerpo. Hay dos tipos de agregación: compartida y compuesta. La agregación compartida es cuando las partes se pueden compartir entre todos y la agregación compuesta es cuando las partes pertenecen al todo. Por ejemplo, la malla (modelo de polígono 3D) que describe la forma de un auto de carreras y está texturizada y representada en una pantalla puede ser compartida por muchos autos de carreras. Como resultado, esto se puede representar como agregación compartida, que se denota con un diamante hueco. Ver figura



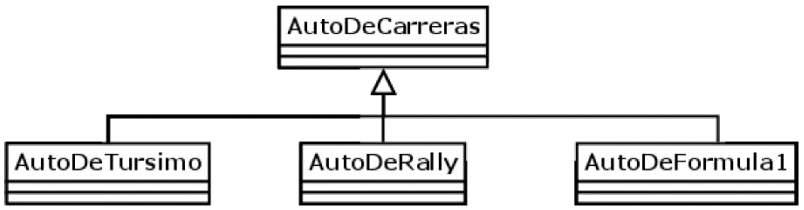
La relación entre una Malla y un AutoDeCarreras se muestra como una agregación compartida.
Tenga en cuenta que la agregación compartida implica que cuando se destruye un AutoDeCarreras, su Malla no se destruye. (Observe también cómo el diagrama muestra que un objeto Malla no sabe nada sobre un objeto AutoDeCarreras).
La agregación compuesta es una relación mucho más fuerte e implica que las partes viven y mueren con el todo. Siguiendo nuestro ejemplo de AutoDeCarreras, podríamos decir que un Chasis tiene este tipo de relación con un automóvil. Un Chassis es propiedad total de un AutoDeCarreras y se destruye cuando se destruye el auto. Este tipo de relación se indica con un diamante relleno como se muestra en la Figura:



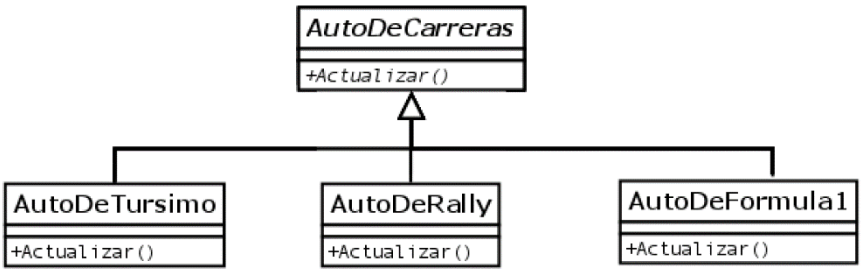
Existe una diferencia muy sutil entre la agregación compartida y la asociación. Por ejemplo, en el diseño discutido hasta ahora, la relación entre un espectador y un auto de carreras se ha mostrado como una asociación, pero como muchos espectadores diferentes pueden ver el mismo coche, podría pensar que está bien mostrar la relación como una agregación compartida. Sin embargo, un espectador no es parte del todo de un coche de carreras y, por tanto, la relación es asociación, no agregación.

Generalización

La generalización es una forma de describir la relación entre clases que tienen propiedades comunes. La generalización describe una relación de herencia. Por ejemplo, un diseño puede requerir que diferentes tipos de AutoDeCarreras sean subclasificados de AutoDeCarreras para proporcionar vehículos para tipos específicos de carreras, como rally, Fórmula Uno o turismo. Este tipo de relación se muestra usando un triángulo hueco en el extremo de la clase base de la asociación como se muestra en la Figura

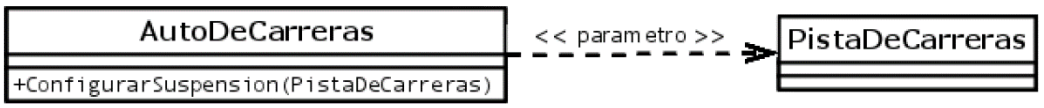


A menudo, en el diseño orientado a objetos usamos el concepto de una clase abstracta para definir una interfaz que se implementará por cualesquiera subclases. Esto es descrito explícitamente por UML usando texto en cursiva para el nombre de la clase y para cualquiera de sus operaciones abstractas. Por lo tanto, si AutoDeCarreras se va a implementar como una clase abstracta con una actualización de método virtual puro, la relación entre este y otros autos de carreras se muestra como en la Figura



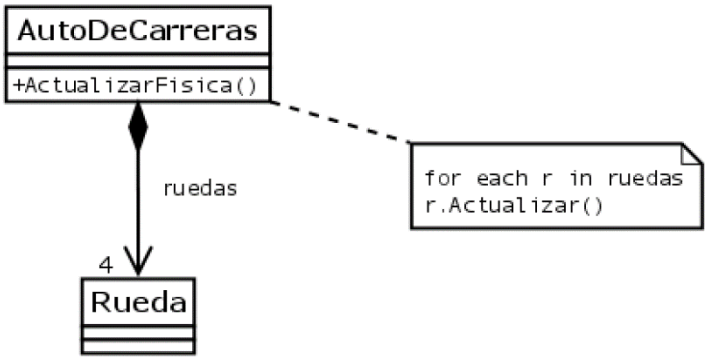
Dependencias

A menudo, encontrará que una clase depende de otra por alguna razón, sin embargo, la relación entre ellas no es una asociación (como la define UML). Esto ocurre por todo tipo de razones. Por ejemplo, si un método de clase A toma una referencia a la clase B como parámetro, entonces A tiene una dependencia de B. Otro buen ejemplo de dependencia es cuando A envía un mensaje a B a través de un tercero, lo que sería típico de un diseño que incorpora el manejo de eventos.
Una dependencia se muestra mediante una línea discontinua con una flecha en un extremo y, opcionalmente, puede calificarse con una etiqueta. La Figura muestra cómo un AutoDeCarreras depende de un PistaDeCarreras.



Notas

Las notas son una característica adicional que puede utilizar para ampliar funciones específicas que necesitan una explicación más detallada. Por ejemplo, si utiliza notas en los diagramas de clases impresos en este apartado para agregar pseudocódigo donde sea necesario. Una nota se representa como un rectángulo con una esquina "doblada" y tiene una línea discontinua que la conecta con el área de interés. La Figura muestra cómo se usa una nota para explicar cómo el método ActualizarFisica de un AutoDeCarreras itera a través de sus cuatro ruedas, llamando al método Actualizar de cada una.



4.3.1.5. Colaboración y composición en un diagrama de clases

Diagrama de colaboración UML

El diagrama de colaboración se utiliza para mostrar la relación entre los objetos de un sistema. Tanto el diagrama de secuencia como el de colaboración representan la misma información, pero de manera diferente. En lugar de mostrar el flujo de mensajes, describe la arquitectura del objeto que reside en el sistema, ya que se basa en la programación orientada a objetos. Un objeto consta de varias características. Varios objetos presentes en el sistema están conectados entre sí. El diagrama de colaboración, que también se conoce como diagrama de comunicación, se utiliza para representar la arquitectura del objeto en el sistema.

Notaciones de un diagrama de colaboración

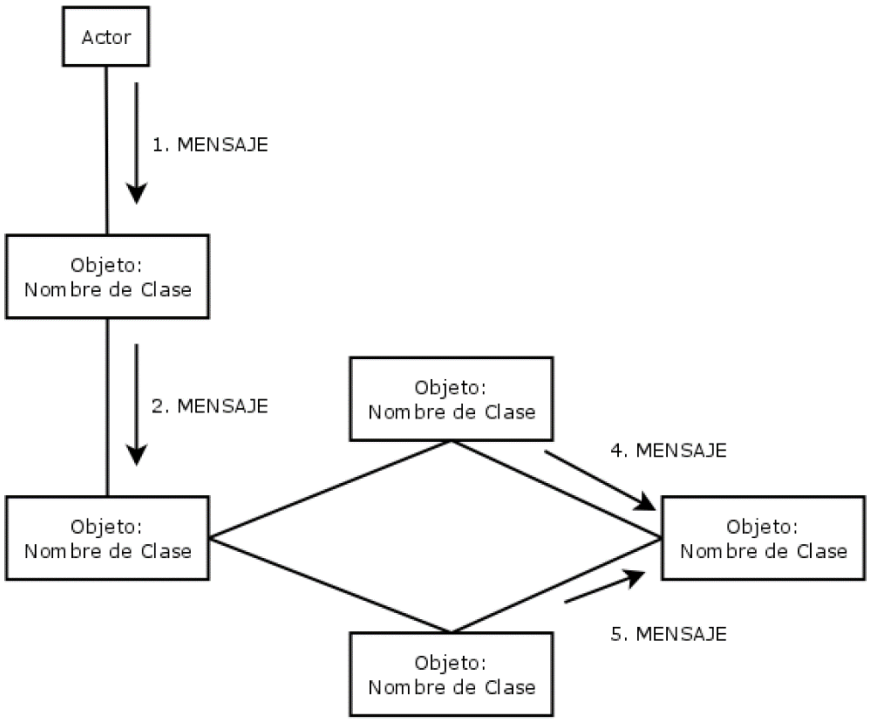
Los siguientes son componentes de un diagrama de colaboración y se enumeran a continuación:

- Objetos: La representación de un objeto se realiza mediante un símbolo de objeto con su nombre y clase subrayados, separados por dos puntos. En el diagrama de colaboración, los objetos se utilizan de las siguientes formas:
  - El objeto se representa especificando su nombre y clase.
  - No es obligatorio que aparezcan todas las clases.
  - Una clase puede constituir más de un objeto.
  - En el diagrama de colaboración, primero se crea el objeto y luego se especifica su clase.
  - Para diferenciar un objeto de otro, es necesario nombrarlos.
- Actores: En el diagrama de colaboración, el actor juega el papel principal, ya que, invoca la interacción. Cada actor tiene su rol y nombre respectivos. En esto, un actor inicia el caso de uso.
- Vínculos: El vínculo es una instancia de asociación, que asocia los objetos y los actores. Representa una relación entre los objetos a través de los cuales se envían los mensajes. Está representado por una línea continua. El vínculo

ayuda a un objeto a conectarse o navegar hacia otro objeto, de modo que los flujos de mensajes se adjuntan a los vínculos.

- Mensajes: Es una comunicación entre objetos que lleva información e incluye un número de secuencia, para que la actividad pueda tener lugar. Está representado por una flecha etiquetada, que se coloca cerca de un enlace. Los mensajes se envían del remitente al receptor, y la dirección debe ser navegable en esa dirección en particular. El receptor debe comprender el mensaje.

Componentes de un diagrama de colaboración



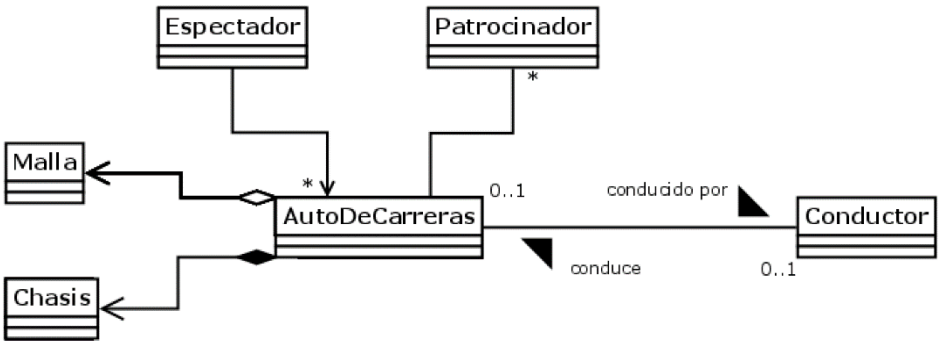
¿Cuándo usar un diagrama de colaboración?

Las colaboraciones se utilizan cuando es imprescindible representar la relación entre el objeto. Tanto el diagrama de secuencia como el de colaboración representan la misma información, pero la forma de representarla es bastante diferente. Los diagramas de colaboración son los más adecuados para analizar casos de uso.
A continuación, se muestran algunos de los casos para los que se implementa el diagrama de colaboración:

- Modelar la colaboración entre los objetos o roles que llevan las funcionalidades de casos de uso y operaciones.
- Modelar el mecanismo dentro del diseño arquitectónico del sistema.
- Capturar las interacciones que representan el flujo de mensajes entre los objetos y los roles dentro de la colaboración.
- Modelar diferentes escenarios dentro del caso de uso u operación, lo que implica la colaboración de varios objetos e interacciones.
- Apoyar la identificación de objetos que participan en el caso de uso.
- En el diagrama de colaboración, cada mensaje constituye un número de secuencia, de modo que el mensaje de nivel superior se marca como uno y así sucesivamente. Los mensajes enviados durante la misma llamada se denotan con el mismo prefijo decimal, pero con diferentes sufijos de 1, 2, etc. según su ocurrencia.

4.3.1.6. Leyendo un diagrama de clases

A estas alturas usted debería estar en condiciones de leer un diagrama como el que se mostró con anterioridad en este capítulo.



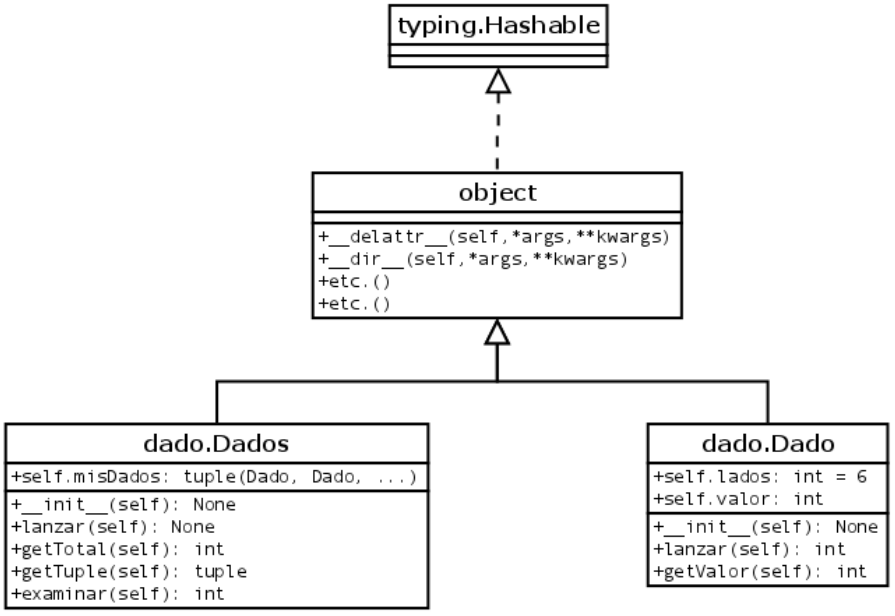
Téngalo en cuenta para la sección de ejercicios.

4.3.1.7. Escribiendo un diagrama de clases

Para mostrar un ejemplo simple utilizaremos un ejemplo de la clase anterior:

```
class Dado(object):
    """Simular un dado genérico."""

    def __init__( self ):
    self.lados = 6
    self.lanzar()

   def lanzar( self ):
       """lanzar() -> number
       Actualiza el dado con un lanzamiento al azar."""
       self.valor = 1 + random.randrange(self.lados)
       return self.valor

    def getValor( self ):
       """getValor() -> number
       Return the last value set by lanzar()."""
       return self.valor

class Dados( object ):
    """Simular un par de dados."""

    def __init__( self ):
        "Crea los dos objetos Dado."
        self.misDados = ( Dado(), Dado() )

    def lanzar( self ):
        "Retorna un numero al azar al lanzar los dados."
        for d in self.misDados:
            d.lanzar()
```

```
def getTotal( self ):
    "Retorna el total para dos dados."
    t= 0
    for d in self.misDados:
        t += d.getValor()
    return t

def getTuple( self ):
    "Return una tupla con el valor de los dados."
    return tuple( [d.getValor() for d in self.misDados] )

def examinar( self ):
    "Retorna True si examinar el valor de cada objeto Dado
    comprueba que son iguales"
    return (self.misDados[0].getValor() ==
  self.misDados[1].getValor())
```



Logramos apreciar que ambos **Dado** y **Dados** heredan de objeto, es decir, comparten ciertas características (la herencia será vista en mayor profundidad en la siguiente clase). Como vimos la relación es una "Generalización". El objeto **object** en Python tiene muchos métodos por lo mismo solo escribimos unos cuantos, si le interesa saber más puede ver el archivo **builtins.py** que es propio de Python y que en el fondo describe la interfaz que debe implementar el objeto. Además, si seguimos indagando apreciamos que object debe implementar la interfaz **typing.Hashable**. Ahora bien, **object** y **typing.Hashable** pueden ser seguidas inspeccionando el código, pero tal inspección está totalmente fuera del alcance de este curso, ya que, implica examinar Python internamente, algo que, aunque es interesante, es de poca utilidad para nuestros propósitos.

4.3.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Carlos Fontela, UML - Modelado de Software para Profesionales, 2011

[5] Python Tutorial.
https://www.w3schools.com/python/