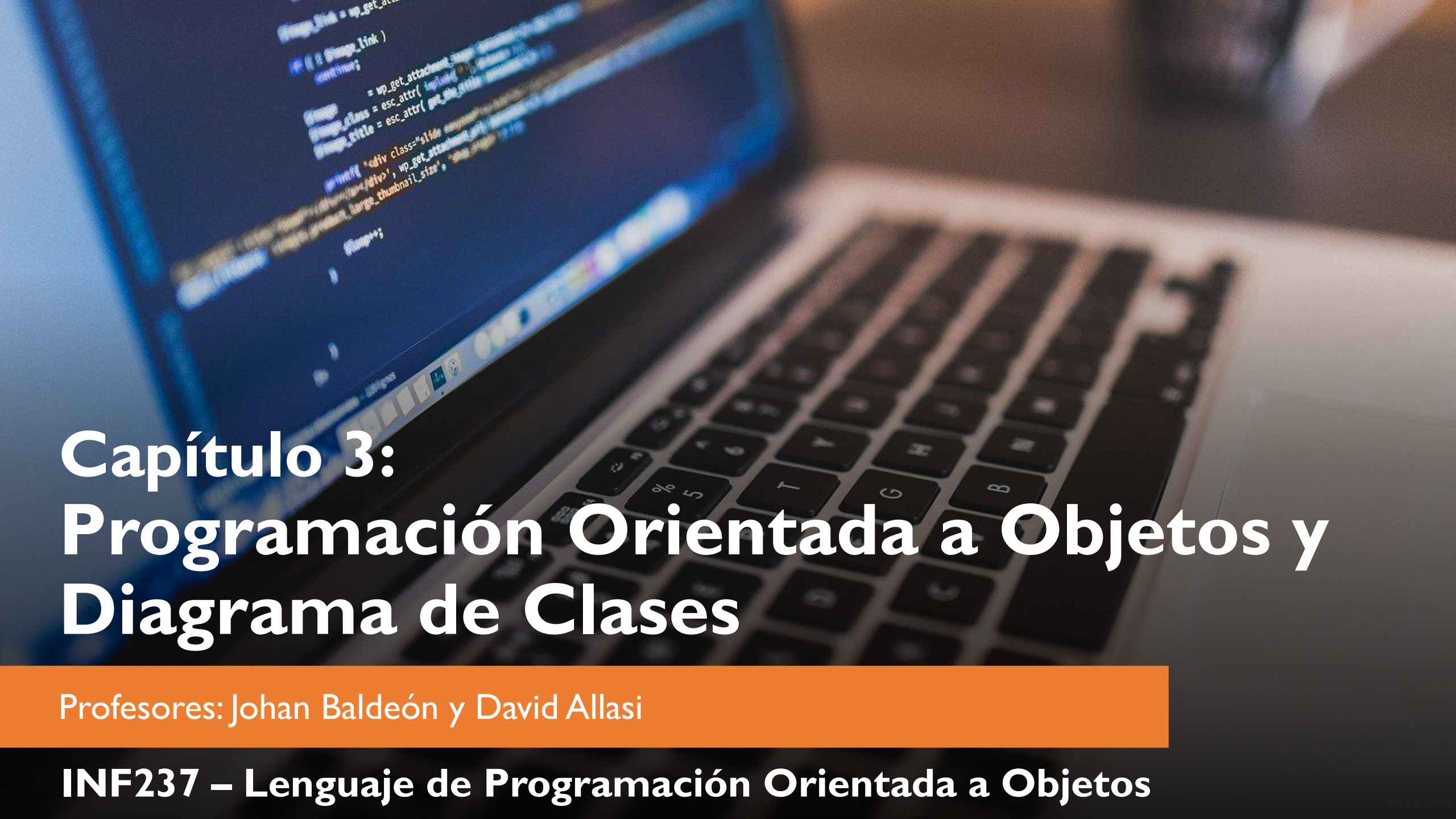


Capítulo 3: Programación Orientada a Objetos y Diagrama de Clases

Profesores: Johan Baldeón y David Allasi

INF237 – Lenguaje de Programación Orientada a Objetos

Agenda

- Diagrama de Clases
  - Clases, atributos, métodos.
  - Asociación.
  - Multiplicidad.
  - Clase asociativa, asociación reflexiva.
  - Composición.
  - Agregación.
  - Herencia
- Caso - Sistema de Ventas

Diagrama de Clases

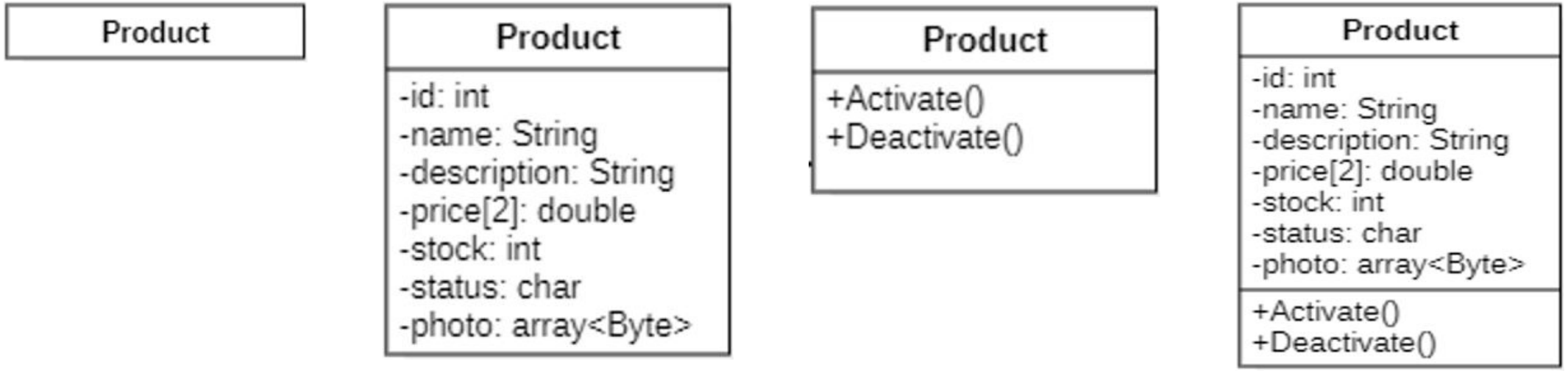
- Un diagrama de clases es un modelo del problema, que captura todos los requerimientos del usuario. Este conocimiento de los requerimientos está contenido en las clases, atributos, métodos y asociaciones entre las clases.

¿Qué información lleva el Diagrama de Clases?

- Las clases representan las entidades físicas y conceptuales del negocio.
- Los atributos representan la información que es conocida por las clases.
- Los métodos definen cómo funcionan los requerimientos del sistema.
- Las asociaciones representan las relaciones entre las entidades del negocio. Ellas capturan las reglas del mismo y permiten la comunicación entre las clases.

Clases

- Una clase es mostrada como un rectángulo con tres partes:
  - Nombre de la clase en la parte superior (Obligatoria)
  - Lista de atributos en la parte central (Opcional)
  - Lista de métodos en la parte inferior (Opcional)
- Tanto los atributos como los métodos pueden ser suprimidos individualmente.
- Los atributos y métodos pueden ser definidos para ayudar en el entendimiento de la clase.

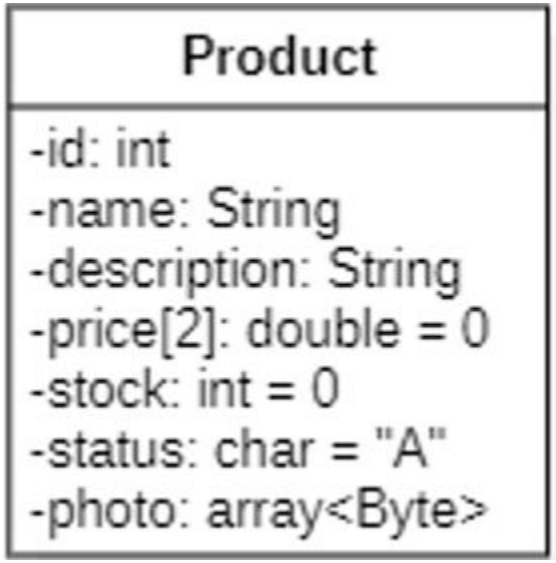


Definición de atributos

- Un atributo está definido por un nombre, un tipo y un valor inicial. (El tipo y el valor inicial son opcionales y dependientes del lenguaje de programación).
- Un atributo indica que un objeto es responsable de conocer una cierta pieza de información.
- La información de un atributo es el menor nivel de granularidad en un objeto.
- Las características de los atributos son opcionales en un diagrama de clases. Se pueden omitir por completo o solo mostrar sus nombres.
- El tipo definitivo del atributo deber ser definido en la etapa de implementación. (Incluso en el diagrama de clases se puede tener tipos genéricos como color, hora, día, etc.)
- Cuando hay clases con un solo atributo, verificar si el atributo no aplica mejor a otra clase.

Notación para los atributos

- "Nombre del atributo"[Multiplicidad]: Tipo = "Valor Inicial"
- El nombre del atributo debe escribirse en minúsculas.
- La multiplicidad indica el número de atributos por cada instancia de la clase. Si no se indica, se asume uno.



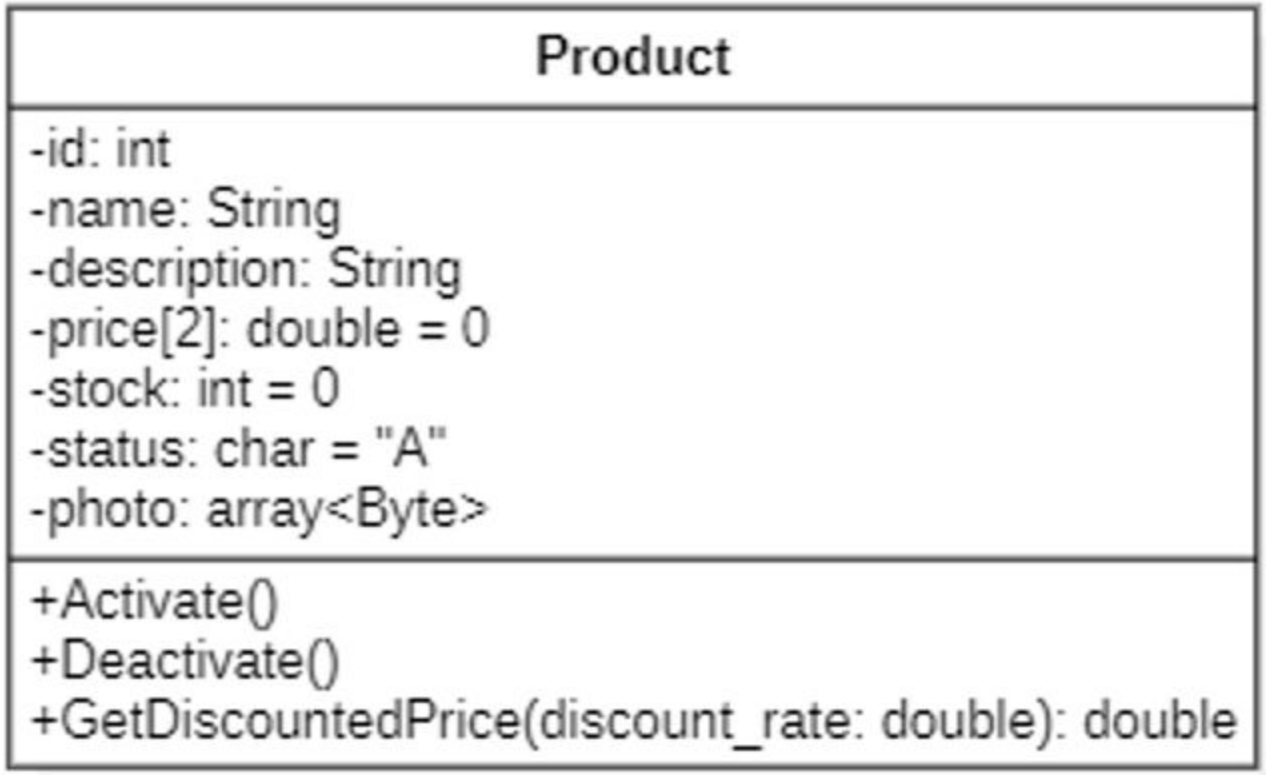
Definición de métodos

- Un método está definido por un nombre, una lista de parámetros y un tipo de retorno.
  - Los parámetros están definidos por un nombre, un tipo y un valor por omisión.
  - Los tipos y los valores por omisión son dependientes del lenguaje de programación.
- Un método indica lo que un objeto es responsable de proveer al sistema funcionalmente mientras que los parámetros muestran la información que es pasada al objeto para ejecutar el método.
- El tipo de retorno muestra la información que es devuelta al que invocó el método.
- Los parámetros así como los tipo de retorno se pueden mostrar opcionalmente en el diagrama de clases.

Notación de métodos

- “nombre del método” (“lista de parámetros”):“tipo de retorno”
- El nombre del método debe ser único para la clase contenedora.
- Lista de Parámetros:
  - Tiene la siguiente estructura:“clase de parámetro” “nombre de parámetro” :“tipo de parámetro” = “valor por defecto”
- El tipo de retorno indica el tipo de dato (un valor o un objeto) que retorna el método al final de su ejecución.

Notación de métodos (2)

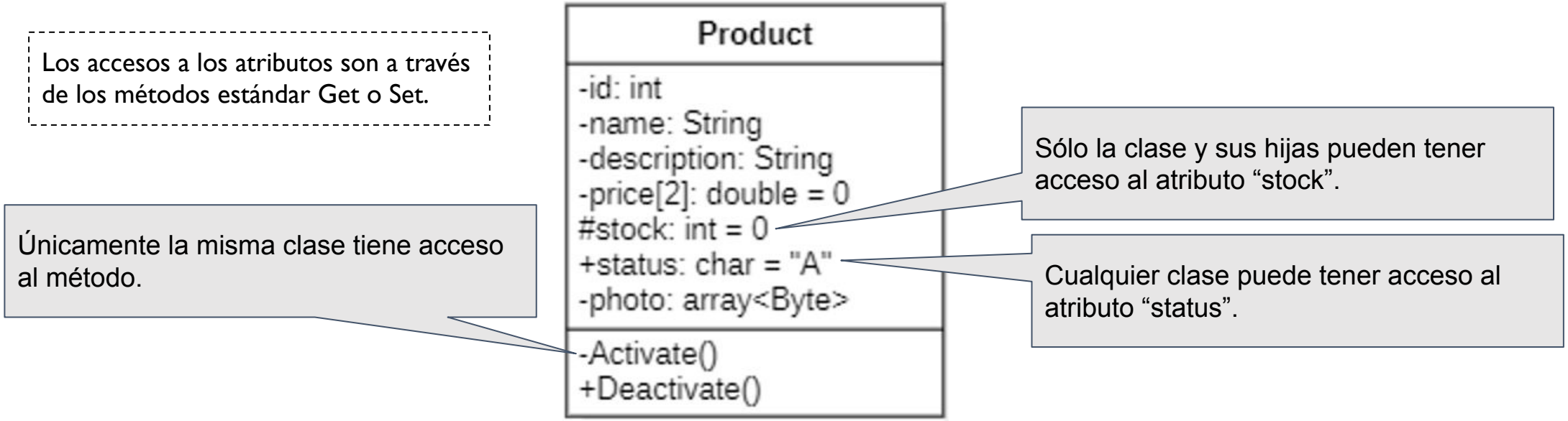


Métodos estándares

- Son métodos comunes a todas las clases.
- Normalmente no se indican en el diagrama de clases.
- Los métodos estándares a considerar son:
  - Constructor (Create): Crea un nuevo objeto de la clase e inicializa su estado.
  - Destructor (Destroy): Destruye el objeto específico y libera su estado.
  - Modificador (Set): Modifica el valor de los atributos de un objeto.
  - Selector (Get): Permite acceder a los valores de los atributos de un objeto.

Visibilidad de atributos y métodos

- Se puede asignar distintos tipos de visibilidad a los atributos y métodos de las clases
- Tipos de visibilidad:
  - pública (+): cualquier clase puede utilizar el método
  - protegida (#): solamente las clases que son especializaciones (descendientes) pueden utilizarlo
  - privada(-): solamente la misma clase puede utilizarla



Notas y restricciones

En ciertas ocasiones es conveniente definir ciertas notas o restricciones en el diagrama de clases.

- Las notas:
  - Se representan dentro de cajas.
  - No son parte del modelo pero si son parte del diagrama de clases.
- Las restricciones (constraints):
  - A diferencia de las notas son elementos del modelo.
  - Se representan dentro de una caja en donde el texto debe estar encerrado entre {}
- Si es requerido se puede usar una línea punteada para indicar el sentido.
- Las notas o restricciones pueden ser usadas libremente en todos los diagramas de UML sobre todo en donde los mismos no pueden fácilmente adaptarse para capturar los requerimientos de las restricciones.

Asociaciones entre clases

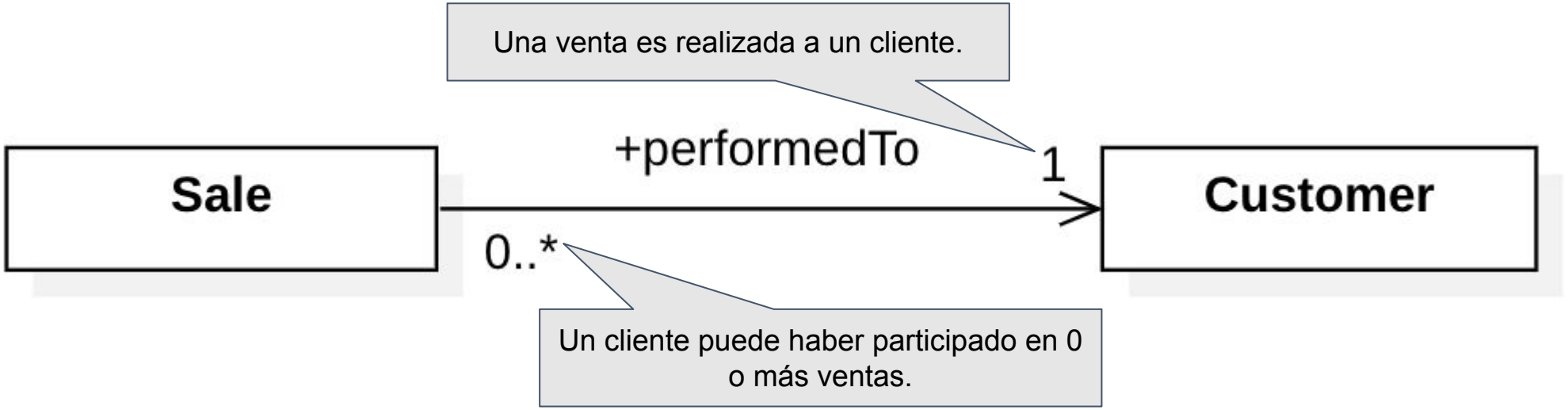
- La Asociación es la relación existente entre dos clases.
- Existen varios tipos de asociaciones:
  - Multiplicidad : Define el número de objetos que pueden estar envueltos en una asociación.
    - Uno a uno,
    - Uno a muchos,
    - Opcional
  - Clase Asociativa : Representa un uso o conocimiento dado a la relación entre objetos.
  - Composición y Agregación: Describen la relación todo y parte entre objetos.
  - Herencia : Es la generalización o especialización de la relación entre dos clases.

Multiplicidad

- La multiplicidad se refiere al número de objetos que son partícipes en una relación.
- En la multiplicidad se examina cada relación y se decide cuántos objetos están envueltos en ella.
- La multiplicidad captura las reglas del negocio en las relaciones, es decir no existe una simple respuesta que siempre sea correcta. Depende enteramente del negocio que se está modelando.
- El criterio de multiplicidad debe ser cierto en todos los puntos del tiempo mientras dure el ciclo de vida del sistema.

Multiplicidad (2)

- Notación:
  - Indicador de multiplicidad es leído de un objeto a otro a través de la asociación utilizando el marcador de multiplicidad.
  - La expresión general de multiplicidad es un rango separado por puntos (donde cada par expresa el mínimo y máximo número de objetos asociados, el "*" indica que no hay límite superior) o un simple número que indica el número exacto de objetos asociados que deben existir.

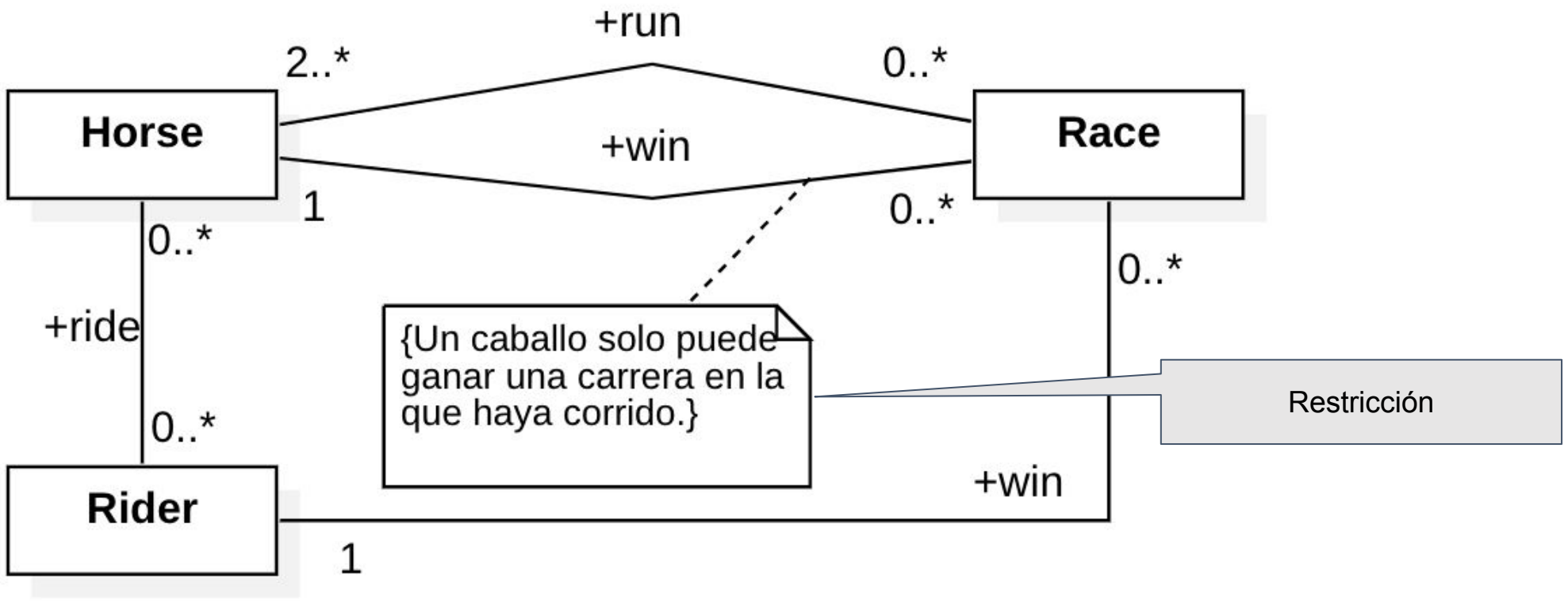


Multiplicidad (3)

- Ejemplo:
  – En un hipódromo se desea implementar un sistema que controle para cada jinete los distintos caballos que este ha cabalgado.
  – También es necesario conocer las distintas carreras en las que ha participado cada caballo y las carreras en las que el caballo ha resultado ganador.
  – Desarrollar el diagrama de clases necesario para soportar el caso descrito.

Multiplicidad (4)

Solución al ejemplo:

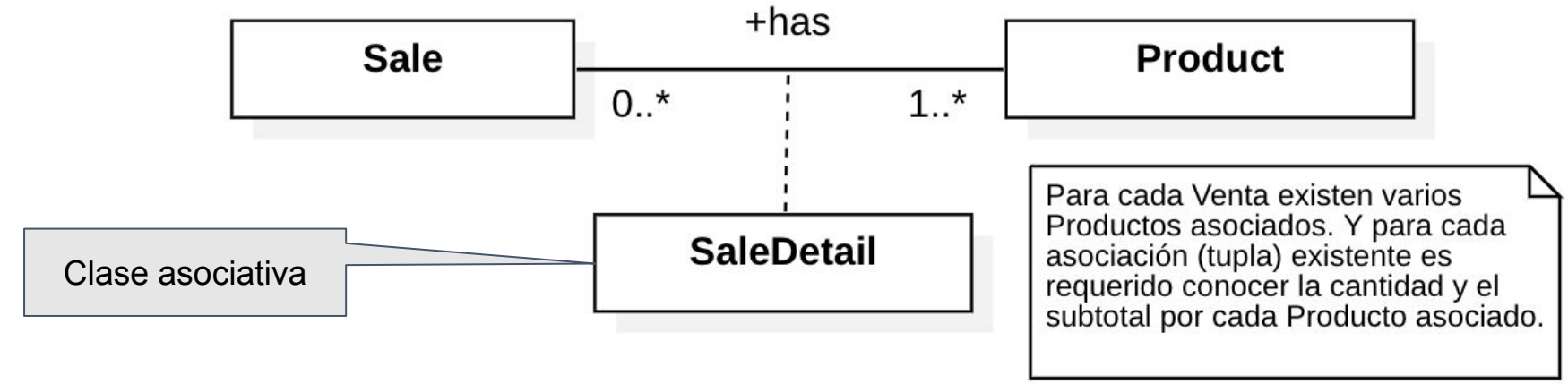


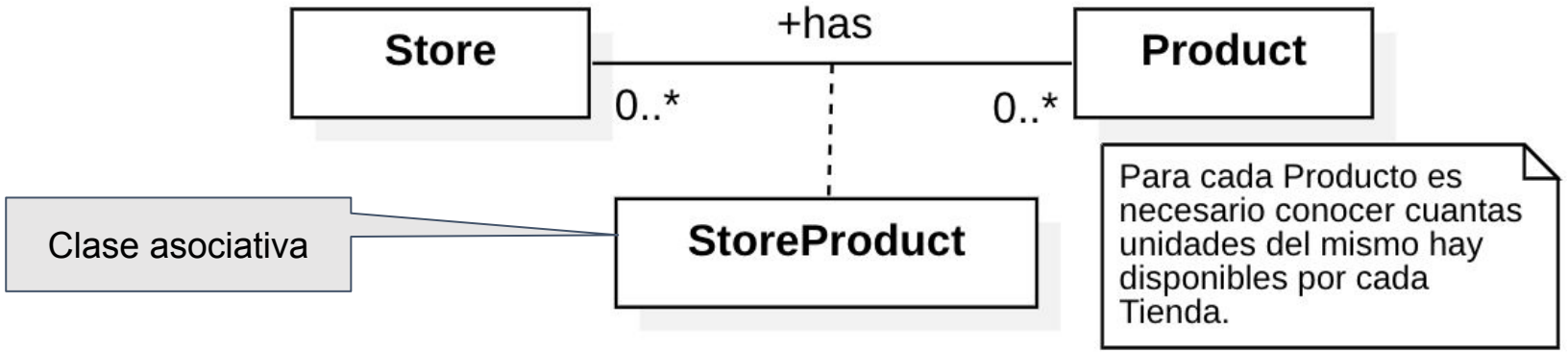
Clase Asociativa

- En algunas ocasiones se determina que ciertos atributos describen características de las relaciones en sí mismas más que pertenecer a un objeto.
- Una clase asociativa es descrita como una restricción de una asociación.
- En los diagramas de clases se deben revisar las asociaciones con multiplicidad muchos a muchos, debido a que normalmente necesitan de una clase asociativa.

Clase Asociativa

Ejemplos de clases asociativas:



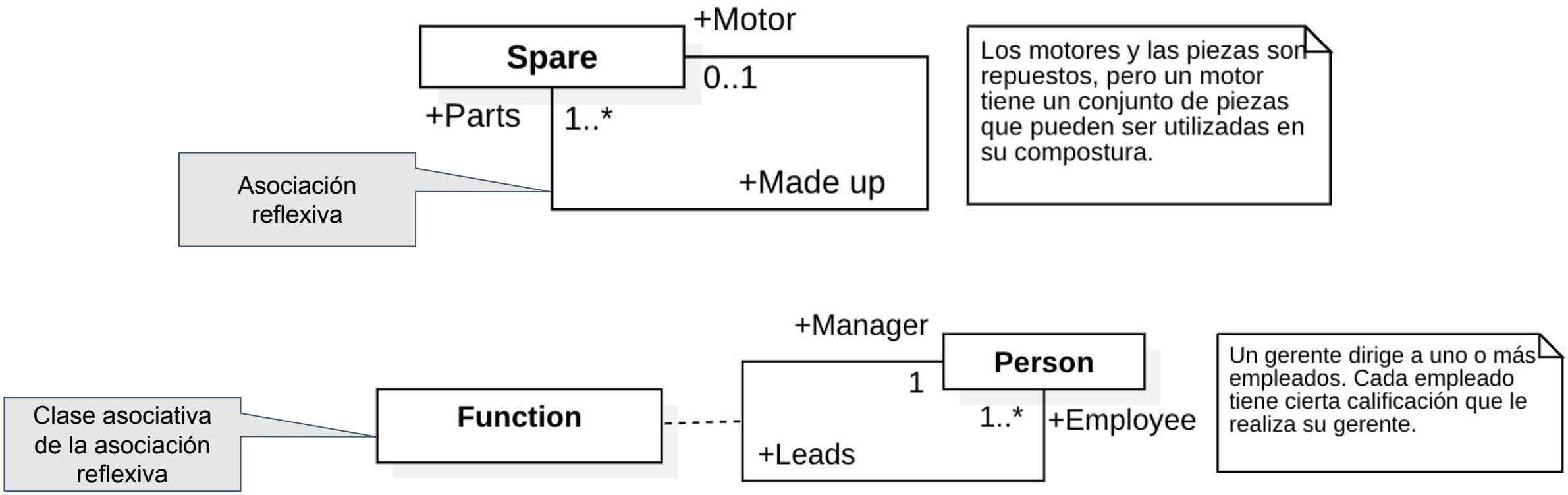


Asociaciones Reflexivas

- Una asociación reflexiva ocurre cuando existe una relación entre dos objetos de la misma clase.
- Los nombres de los roles pueden ser usados para describir la participación de cada uno de los objetos en la relación.
- Las asociaciones reflexivas pueden también tener definidas clases asociativas.

Asociaciones Reflexivas

Ejemplos de asociaciones reflexivas:



Composición

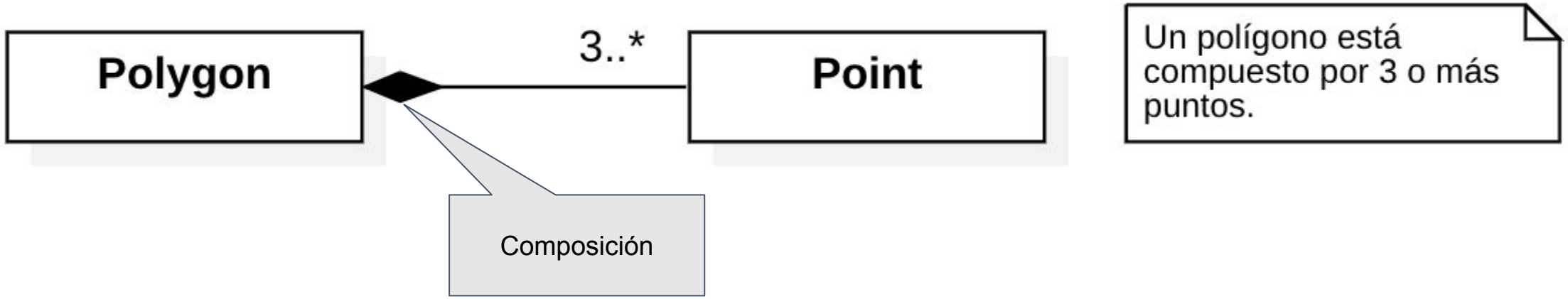
- Una composición es un tipo especial de asociación donde un objeto contiene un número de sub-objetos dentro de él.
- Especifica la relación de un todo con sus partes (Todo Parte).
- Debe entenderse como "consiste de" o "es parte de".
- Características de las relaciones tipo composición
  - El Todo es el propietario de todas las Partes.
  - El Todo es quien crea y destruye todas las Partes.
  - Las Partes son accesadas únicamente a través del Todo.
  - Algunos métodos del Todo ejecutan el mismo método en sus Partes.
- La razón para diferenciar entre una asociación y una composición es que refleja de una mejor manera la forma como nosotros vemos las cosas.

Composición (2)

- Notación:

– En el diagrama de clases de UML, un diamante colocado al lado de la clase designada como el Todo indica una relación tipo Composición.

– Esto define que el Todo está hecho, contiene y/o tiene las partes.



Composición (3)

- Razones para usar una Composición:
  - Las partes pueden ser omitidas del diagrama sin perder el modelo su contenido.
  - El Todo es responsable de administrar todas las Partes.
  - Las partes se desarrollan y pertenecen en la misma unidad del todo.
  - Los mensajes enviados de otros objetos se reciben a través del Todo.
  - Están identificados como un grupo de componentes reusables.
- Razones para no usar una Composición:
  - Al Todo pueden no pertenecer sus Partes.
  - Las Partes pueden existir independientemente del Todo.
  - El Todo no puede crear o eliminar las Partes.
  - Las Partes pueden no morir si el Todo muere.

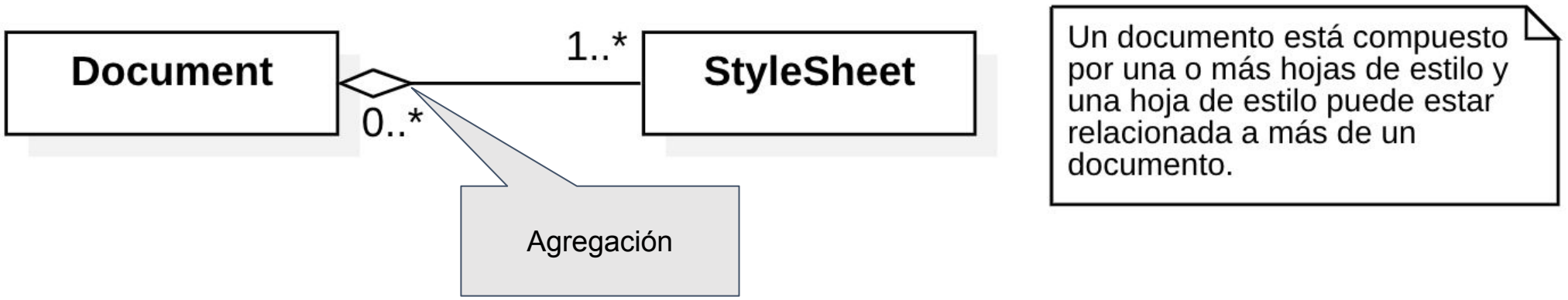
Agregación

- Es otra relación entre objetos del tipo Todo y Parte.
- Se diferencia principalmente de la composición en el sentido que las partes de una agregación no se encuentran tan fuertemente vinculadas al Todo y pueden existir a pesar que el Todo haya sido destruido.
- En este tipo de relaciones es posible además encontrar a una Parte, perteneciendo a la vez a más de un todo.

Agregación

- Notación:

– En el diagrama de clases de UML, un diamante vacío puesto al lado de la clase designada como el Todo indica una relación tipo Agregación.

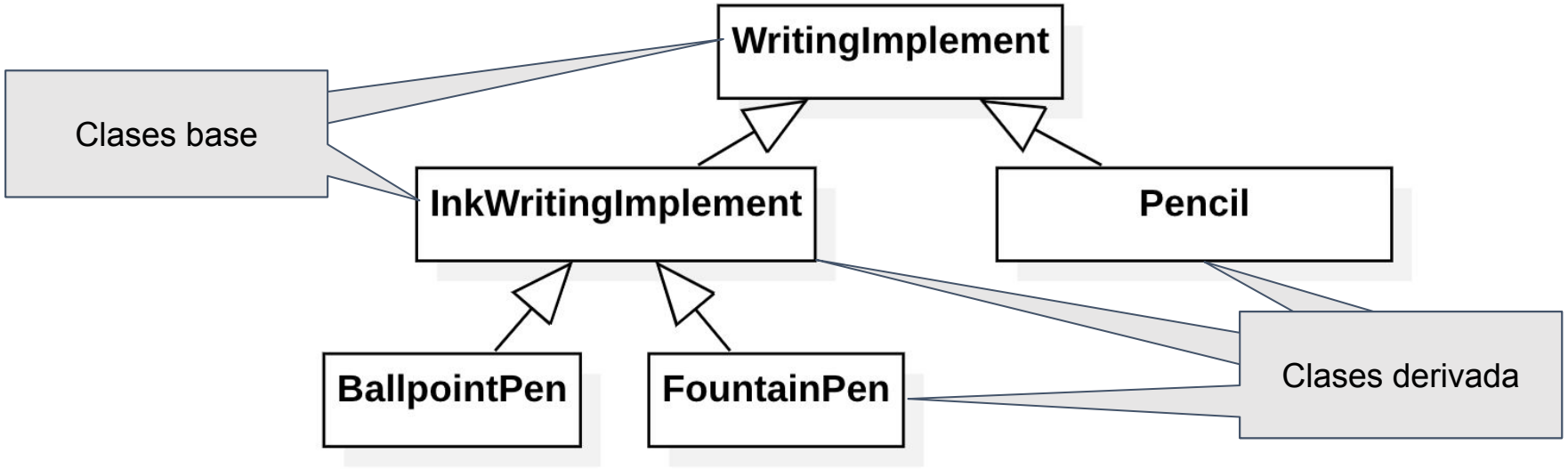


Herencia

- Es la generalización o especialización de una relación entre clases.
- La herencia es una relación entre clases no entre objetos.
- El uso de herencia permite simplificar nuestro entendimiento del sistema.
- Permite adicionar nuevas clases utilizando clases ya existentes, las cuales siguen trabajando exactamente como lo estaban haciendo previamente.
- Es posible especificar un sistema completo sin usar herencia. Pero si es adoptado este criterio se debe utilizar y especificar completamente.

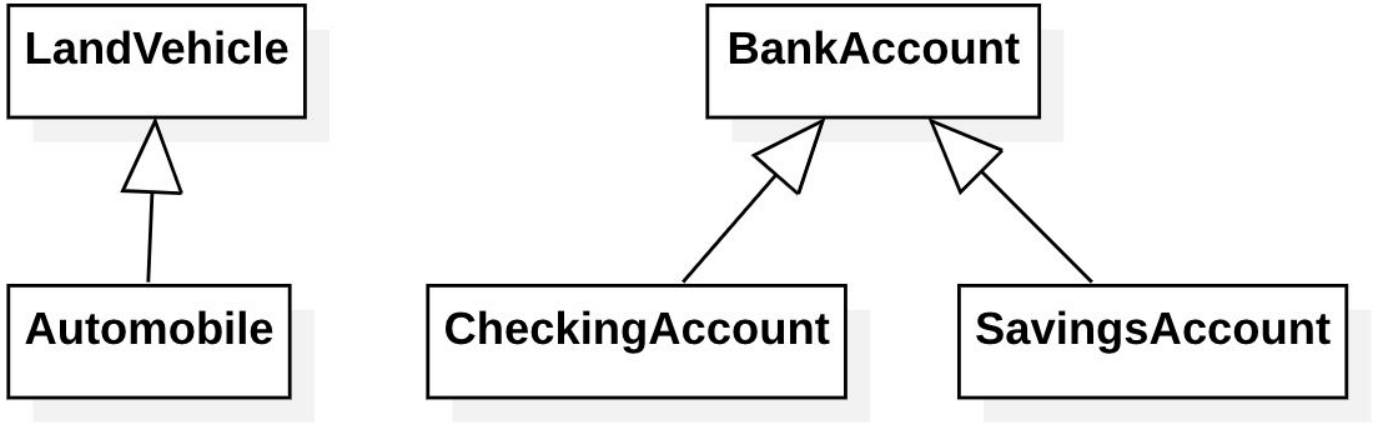
Herencia (2)

- Jerarquía de Herencia:
  - Clase Base (Superclase):
    - Especifica las características generales.
  - Clase Derivada (Subclase):
    - Hereda las características de sus clases padre.
    - Agrega nuevas características.
    - Puede modificar o redefinir métodos de sus clases padre.



Herencia (3)

- Notación:
  - La herencia es especificada como una flecha entre la clase hijo y la clase padre con la punta apuntando a la clase padre.
  - Esta notación indica que cuando una subclase hereda de una superclase, la subclase adquiere automáticamente todas las características de la superclase.
    - Todos los atributos
    - Todos los métodos.
    - Todas las asociaciones.



Herencia (4)

- Especialización:
  - Mientras dure la etapa de análisis, constantemente se descubren nuevas clases a adicionar al diagrama de clases.
  - Durante este proceso de descubrimiento algunas de estas clases pueden ser muy parecidas a otras clases existentes. Teniendo estas nuevas clases algunos atributos u métodos adicionales.
  - Estas nuevas clases deben ser definidas como una subclase de una clase existente. Se puede decir que son clases especializadas.
  - Con la especialización se puede:
    - Reusar servicios desarrollados y probados en otras clases.
    - Desarrollar aplicaciones más rápidamente y con un mayor nivel de confianza.
    - Generar una implosión de código en la medida que el código de una superclase sea usado por varias subclases.

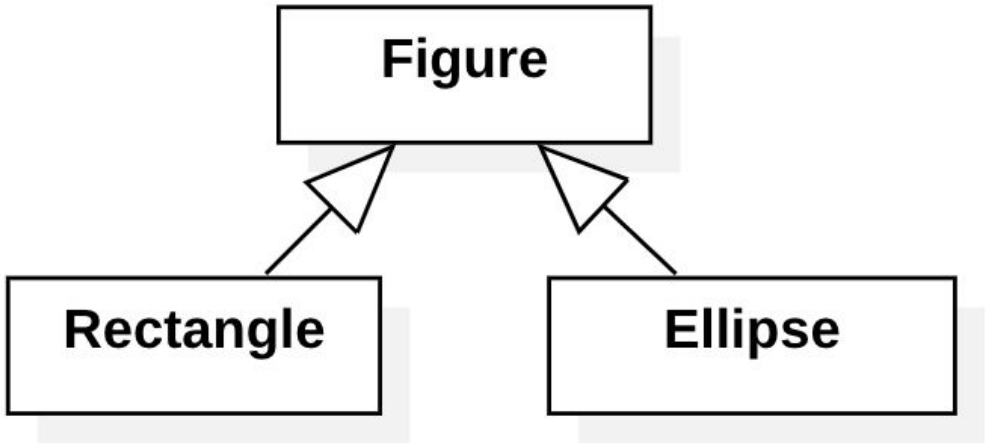
Herencia (5)

- Generalización:

– Es el proceso de reunir características comunes de varias clases en una superclase.

– Permite reducir el código duplicado entre clases similares.

– Genera un modelo más realista.



A pesar de que las clases Rectángulo y Elipse son muy diferentes comparten atributos y métodos similares como ubicación, color, tamaño, etc. lo que nos permite crear una superclase Figura que contenga los atributos, métodos y asociaciones comunes a ambas clases.

Herencia (6)

- Clases Abstractas:
  - Una clase abstracta es aquella que nunca tendrá instancias directas creadas.
  - La clase solo existe en la jerarquía de clases debido a lo cual otras clases pueden heredar de ella.
  - Frecuentemente estas clases son producto de la generalización de otras clases.

- Clases Concretas:
  - Son las que representan a los objetos que pueden existir una vez que se inicia la ejecución del sistema modelado.

Herencia (7)

- Decidir cuándo una clase debe ser abstracta o no, depende exclusivamente del dominio del negocio.
- En algunas metodologías consideran todas las hojas de la jerarquía de herencias como clases concretas y el resto como clases abstractas.
- Esto trae como beneficio que si se necesita la modificación de una subclases no se ven afectadas otras subclases.

Herencia (8)

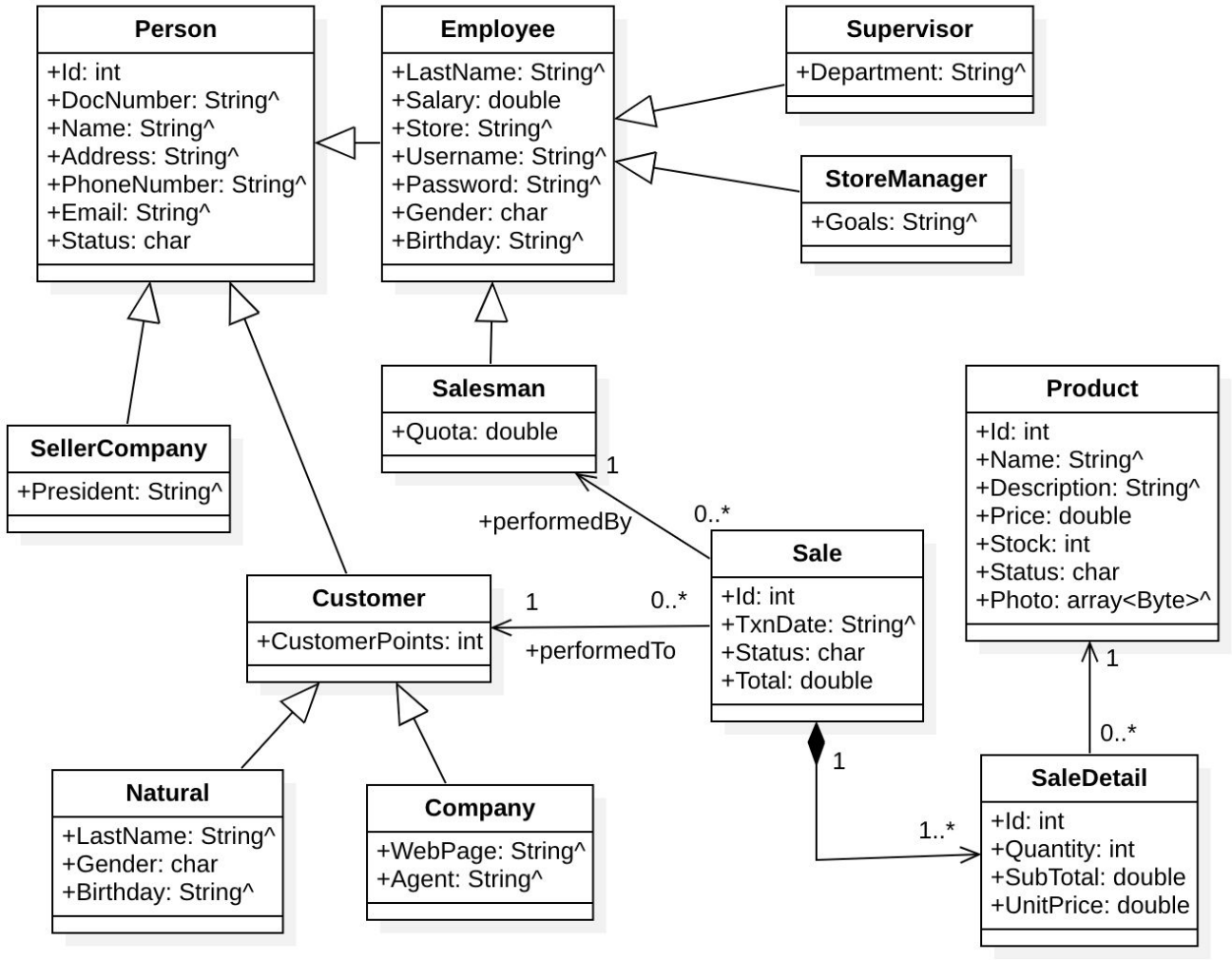
- ¿Cuán profundo debe ser una jerarquía de herencia?
  - La jerarquía de herencia soporta cualquier nivel de generalización.
  - Pero en la práctica se encuentra que hasta cinco niveles de generalización son convenientes.
  - Más allá de esto, la complejidad aumenta. Se hace más complicado entender una sola clase debido a que se necesita conocer todos los niveles previos de la misma, lo que llega a ser muy complicado.

- ¿Cuán ancho debe ser una jerarquía de herencia?
  - No existen límites para el ancho de las jerarquías de herencia.

Caso - Sistema de Ventas

Horario 06M1

Caso Práctico - Sistema de Ventas



Bibliografía

- Ambler, S. W. (2002). *The elements of UML style.* Cambridge, England: Cambridge University Press.
- Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The unified modeling language user guide* (2nd ed.). Boston, MA: Addison-Wesley Educational.
- Fowler, M., & Scott, K. (2003). *UML distilled* (3rd ed.). Boston, MA: Addison-Wesley Educational.
- Larman, C. (2004). *Applying UML and patterns* (3rd ed.). Philadelphia, PA: Prentice Hall.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2005). *The Unified Modeling Language reference manual*. Addison-Wesley Professional.

Bibliografía

- Ragunathan, V. (2016). *C++/CLI primer : for .NET development*. United States New York, NY: Apress,Distributed to the Book trade worldwide by Springer Science+Business Media.
- Templeman, J. (2013). *Microsoft Visual C++/CLI step by step*. Redmond, Wash: Microsoft Press.
- Fraser, S. (2009). *Pro Visual C++/CLI and the .NET 3.5 Platform*. Berkeley, CA New York: Apress Distributed to Book the trade by Springer-Verlag.
- Hogenson, G. (2008). *Foundations of C++/CLI : the Visual C++ Language for .NET 3.5*. Berkeley, CA New York: Apress Distributed to the Book trade by Springer.
- Heege, M. (2007). Expert C++/CLI : .NET for Visual C++ programmers. Berkeley, CA New York: Apress Distributed to the Book trade by Springer-Verlag.
- Sivakumar, N. (2007). *C++ / CLI in action*. Greenwich, CT: Manning.

Bibliografía

- Horton, I. (2014). *Ivor Horton's beginning Visual C++ 2013*. John Wiley & Sons, Wrox.
- Stroustrup, B. (2018). *The C++ programming language* (4th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *A tour of C++*. Pearson Education.
- Stroustrup, B. (2014). *Programming: Principles and Practice Using C++* (2nd ed.). Addison-Wesley.
- Stroustrup, B. (1994). *The design and evolution of C++*. Reading, Mass: Addison-Wesley.