

Patrones Software

Tema 1-1:

Conceptos de POO

y

Diagrama de clases

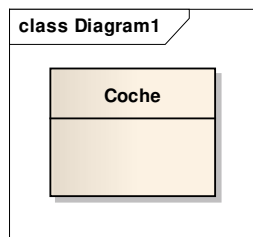
Clases y objetos

- Una clase es un contenedor de datos (atributos) junto con las operaciones para manipular esos datos.
- **Clase = atributos + operaciones**
- Una **clase** es una **plantilla** de la cual se pueden crear **instancias** que serán los **objetos**.
- **Una clase es un tipo de datos y un objeto es una variable de ese tipo.**



Clase

- Una clase se puede representar:
 - como una caja que solo contenga el nombre de la clase
 - como una caja dividida en 3 compartimentos que contendrán:
 - El nombre de la clase
 - Los atributos (Una clase puede tener varios o ningún atributo)
 - Las operaciones o métodos



Clase

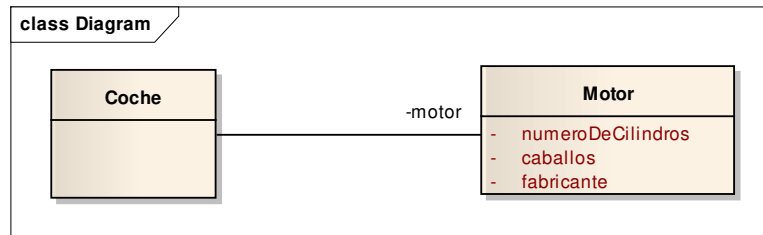
- **Atributos**
 - Un **atributo** es una propiedad o **característica de una clase**.
 - Todo objeto de la clase tiene un valor específico en cada atributo.
 - **Los atributos representan el estado de un objeto.**
 - Una clase puede tener varios o ningún atributo.
- **Operaciones**
 - Las operaciones permiten manipular los datos.



Clase: Atributos

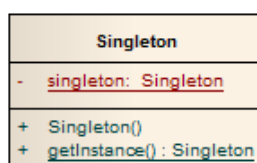
- Notación:

- “En línea”: los atributos se muestran dentro de la clase (notación clásica).
- “Atributos por relación”: los atributos no se muestran dentro de la clase. Esta notación produce un diagrama de clase más grande, pero muestra un mayor detalle para atributos complejos.



Clase: Elementos estáticos

- Son compartidos por todos los objetos de una clase: **todos los objetos de una clase comparten la misma copia de un atributo u operación.**
- Un ejemplo de uso de atributos y operaciones estáticos es el patrón de diseño **Singleton**, que asegura que **solo se podrá construir un objeto de una determinada clase.**
- No es necesario instanciar la clase para poder usarlos, ya que no dependen de la instancia, sino de la clase.
- Para indicar que un atributo u operación es estático se subraya.

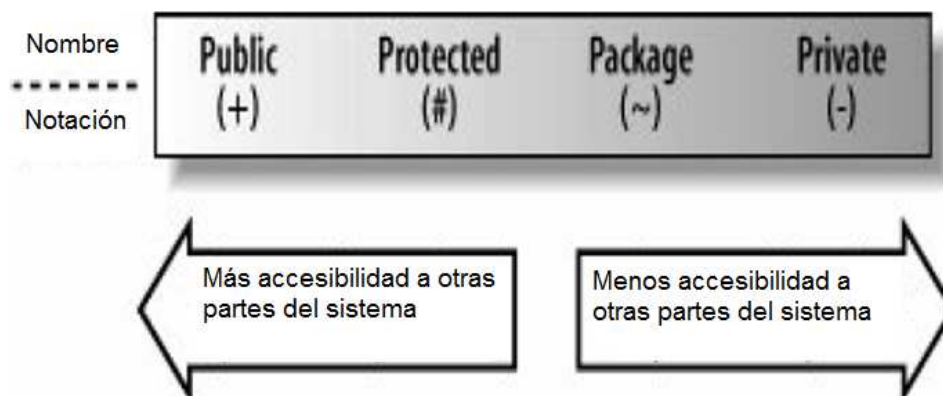


Clase: Modificadores de acceso

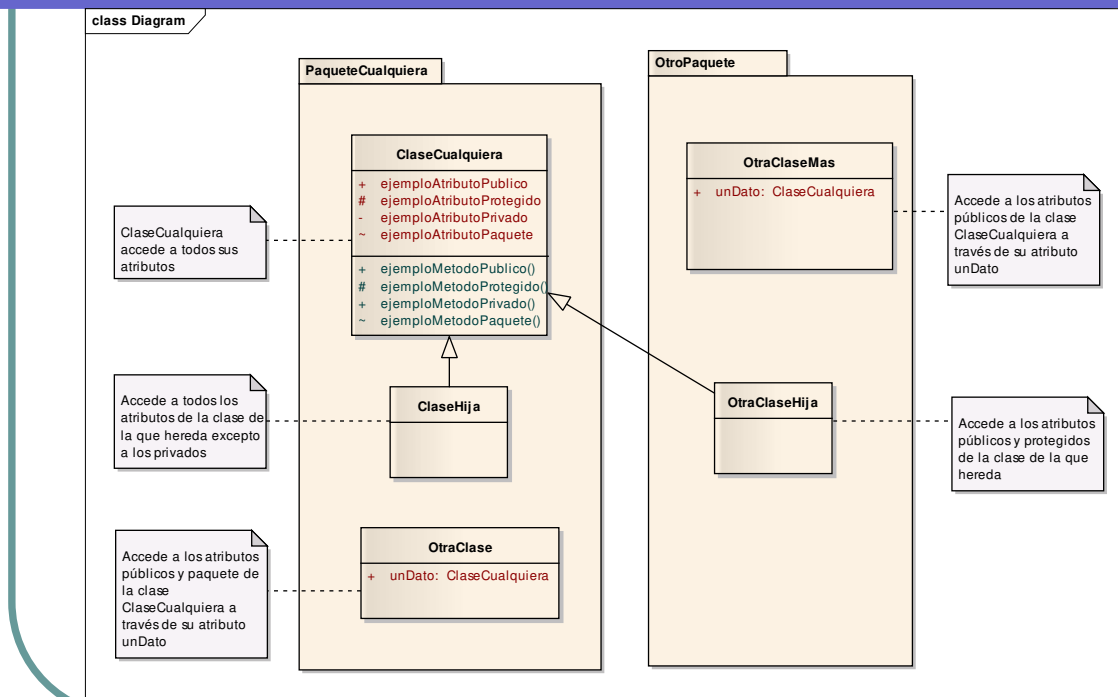
- **Public (+):** Todo el mundo puede acceder al elemento.
Si es un atributo, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo.
Si es un método todo el mundo puede invocarlo.
- **Private (-):** Sólo se puede acceder al elemento desde la propia clase.
- **Protected (#):** es una combinación de los accesos que proporcionan los modificadores public y private. Protected proporciona:
 - acceso público para las clases derivadas
 - acceso privado (prohibido) para el resto de clases
- **Package (~):** se puede acceder al elemento desde cualquier clase del paquete donde se define la clase.



Clase: Visibilidad



Clase: Ejemplo de visibilidad

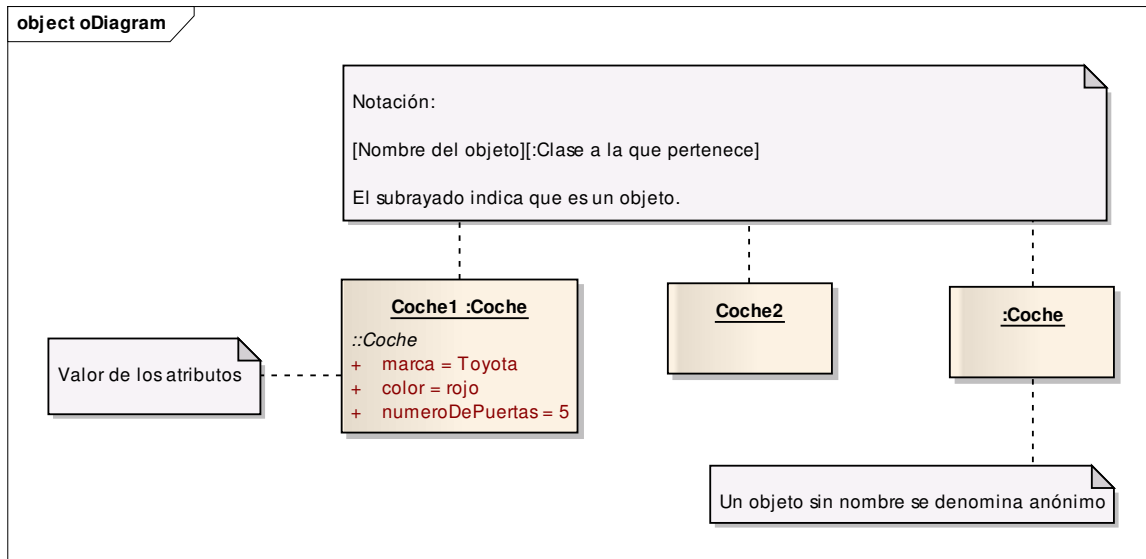


Objetos

- Un **objeto** es una **instancia de una clase**:
 - Con la que se puede **interactuar**: se le pueden enviar mensajes y éste reaccionará ante ellos. Un mensaje es “algo” que le hacemos a un objeto. **Enviar un mensaje = Llamar a un método u operación.**
 - Que tiene **estado**: un objeto lo constituyen todos los datos (**atributos o variables de instancia**) que encapsula en un momento determinado cada uno de los cuales tiene un valor. Los atributos pueden ser constantes o cambiar de valor.
 - Que tiene **comportamiento**: el objeto puede reaccionar de manera diferente a un mensaje en función de su estado.
 - Que tiene **identidad**: al objeto se le hace referencia por un nombre (excepto en los objetos anónimos).



Objetos

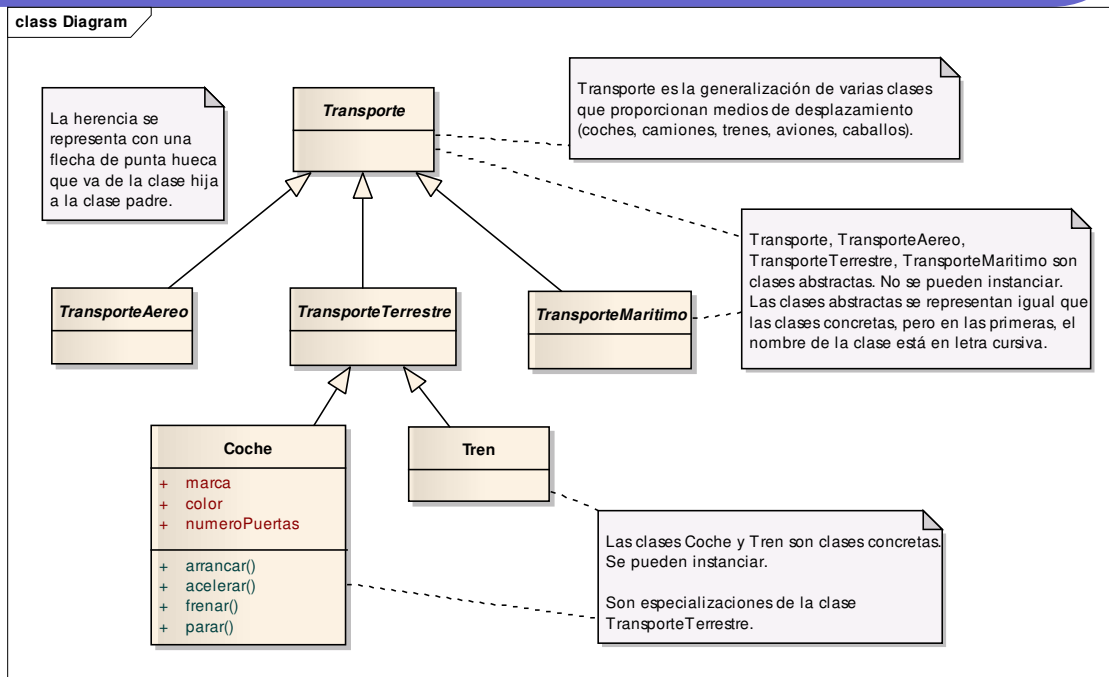


Generalización/Especialización

- **Generalización (clase padre):**
 - Identifica y define los atributos y operaciones comunes en una colección de objetos.
 - Se implementa a través de la herencia.
- **Especialización (clase hija):**
 - Algunas veces necesitamos especializar una clase, modificar la implementación de una parte de su comportamiento.
 - Para personalizar los miembros de una subclase, se deben sobrescribir los miembros de la superclase.
 - La especialización es la herencia con la adición y modificación de métodos para resolver problemas específicos.



Generalización/Especialización



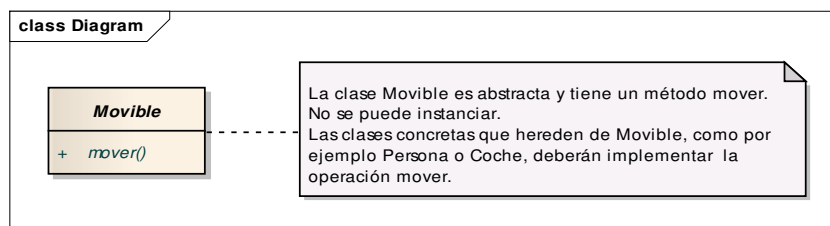
Métodos

- Un **método** es la **implementación** de una **operación**.
- Cada clase provee una implementación para sus operaciones o la hereda de una superclase (**herencia**).
- Si la clase no provee una implementación para una operación y esta tampoco es provista por su superclase, la operación se considera abstracta (**clases abstractas**).



Clases abstractas

- Las **clases abstractas** definen atributos, y operaciones que deben ser implementadas.
- **No se pueden instanciar** ya que falta la implementación de una o varias operaciones.
- **Se utilizan como clase base de otras clases.**
- Son las subclasses las que deben implementar las operaciones.
- Una clase es abstracta si tiene alguna operación abstracta.
- Para indicar que una clase es abstracta se pone el nombre de la clase en cursiva.



Interfaces

- **Una interfaz es una clase abstracta pura**, es decir, ninguna de las operaciones tiene implementación.
- Una interfaz tiene
 - **Atributos** (lenguajes como Java no soportan atributos).
 - Operaciones
- Es un conjunto de operaciones que una clase presenta a otras (contrato).
- Definen un **comportamiento común para varias clases** aunque estas no desciendan de una clase común.
- Son un mecanismo para **implementar herencia múltiple** que evita los problemas derivados de esta.
- Java no permite herencia múltiple, pero si implementar cualquier número de interfaces.



Interfaces

- **Relación interfaz-clase:**

- **Realización:** Es la relación que se produce entre una clase y una interfaz cuando una **clase implementa esa interfaz**. La **clase es proveedora de la interfaz**.

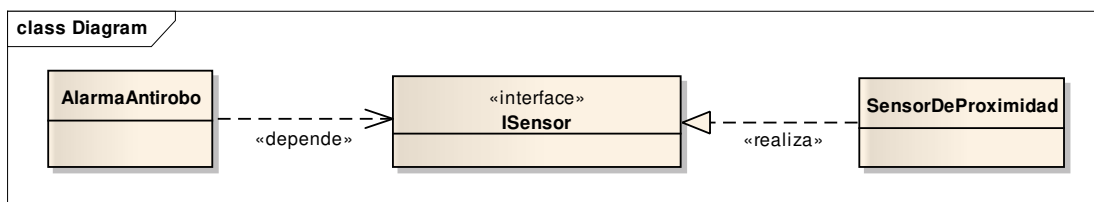
Que una interfaz declare atributos no significa que necesariamente su realización tenga que implementar dicho atributo. Estos pueden aparecer solo como parte de la interfaz para que sean vistos por observadores externos.

- **Dependencia:** Es la relación entre una clase y una interfaz en la que la **clase usa la interfaz**. La clase **requiere** la interfaz.



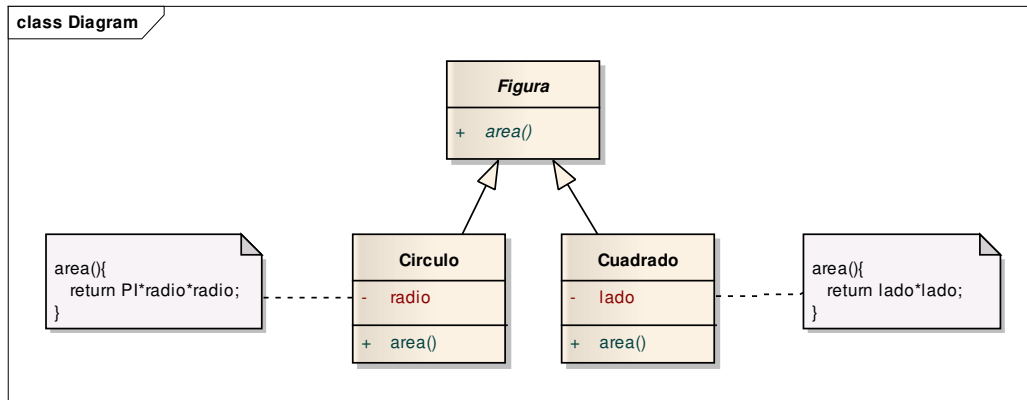
Interfaces

- Notación clásica: Clase estereotipada como <<interface>> y flechas
 - Interfaz ofrecido o provisto: flecha discontinua con punta hueca
 - Interfaz requerido: flecha de dependencia



Polimorfismo

- Característica que permite implementar múltiples formas de un mismo método.
- Permite implementar una operación heredada en una subclase.
- Esto hace que se pueda acceder a una variedad de métodos distintos (todos con el mismo nombre) utilizando exactamente el mismo medio de acceso.

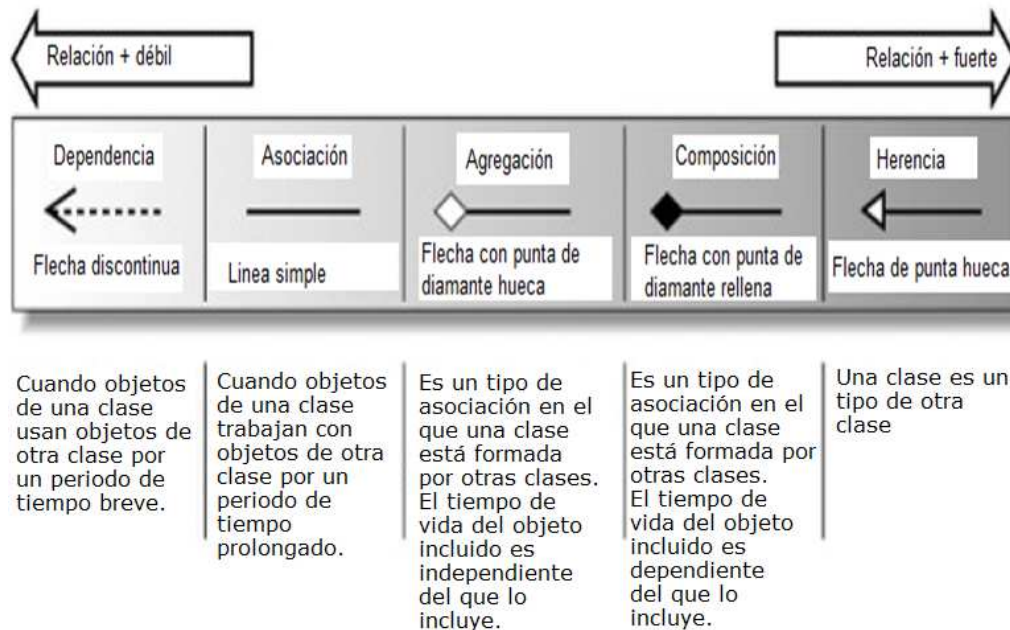


Relaciones entre clases

- Las clases no trabajan solas, sino que se relacionan con otras clases para cumplir con su objetivo.
- Tipos de relaciones de menor a mayor acoplamiento:
 - Dependencia o instanciación
 - Asociación
 - Agregación
 - Composición
 - Herencia

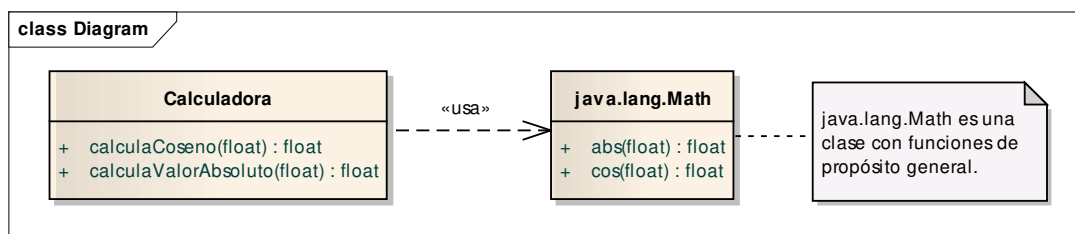


Relaciones entre clases



Dependencia o Instanciación (uso)

- Es un tipo de **relación en el que una clase utiliza a otra**, es decir, **una clase es instanciada por otra**.
- La relación de dependencia se usa a menudo cuando hay clases que proveen un conjunto de funciones de propósito general. Ejemplo: clase `java.lang.Math` de Java.
- La dependencia se representa con una flecha discontinua.

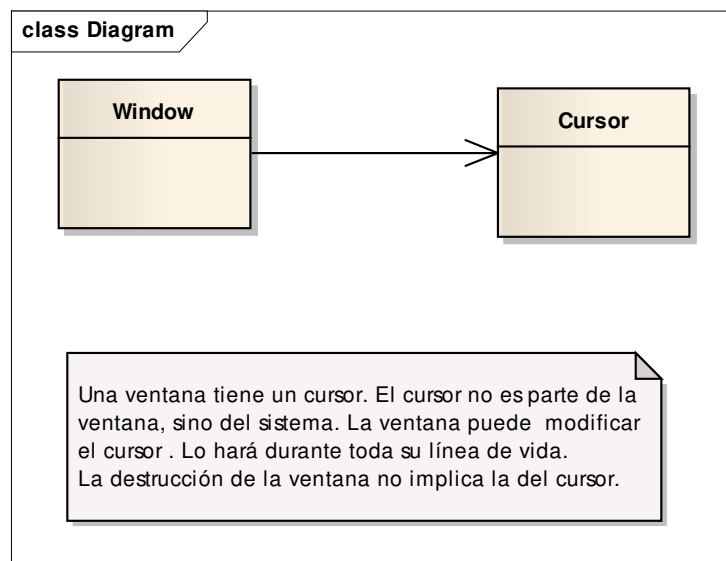


Asociación

- La **asociación** es una relación más fuerte que la **dependencia**.
- Por lo general indica que una clase mantiene una **relación con otra clase durante un período prolongado de tiempo**.
- Las **líneas de vida** de dos objetos vinculados por una asociación probablemente **no están unidas** (la **destrucción de uno no implica la destrucción del otro**).
- En una asociación, una clase contiene una referencia a un objeto u objetos de otra clase en forma de atributos.
- Las asociaciones se suelen leer como “... **tiene un** ...”. No confundir con composición y agregación (que se leen como “...está formado por...”)



Asociación



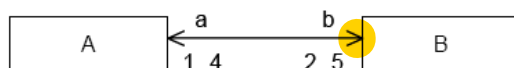
Asociación

- **Características de la asociación:**

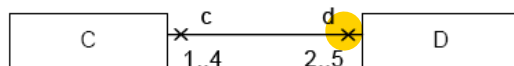
- **Navegabilidad:** se aplica a menudo a una asociación para indicar qué clase contiene el atributo que soporta la relación.
- **Nombre** de la asociación (opcional): frase para indicar el contexto de la asociación.
- **Multiplicidad** de la asociación (opcional): indica la cantidad de objetos de una clase que pueden relacionarse con un objeto de una clase asociada



Asociación Navegabilidad



La asociación es navegable en ambos sentidos. A tendrá un atributo de tipo B y B tendrá un atributo de tipo A.



La asociación no es navegable.



No se especifica la navegabilidad.



La asociación es navegable en un solo sentido. G tendrá un atributo de tipo H.



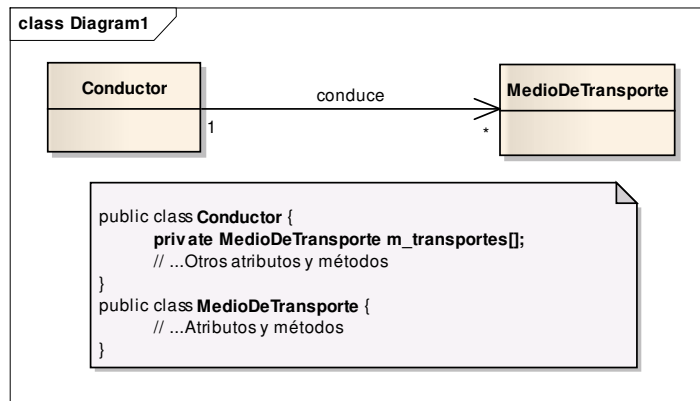
La asociación es navegable en uno de los sentidos y en el otro no se especifica la navegabilidad. I tendrá un atributo de tipo J, J puede tener o no un atributo de tipo I.



Asociación

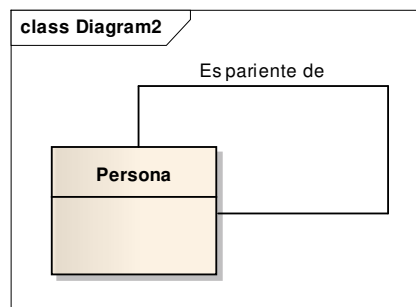
Nombre y multiplicidad

- **Nombre** de la asociación: frase para indicar el contexto de la asociación.
- La **multiplicidad** de la asociación: indica la cantidad de objetos de una clase que pueden relacionarse con un objeto de una clase asociada.



Asociaciones reflexivas

- En algunas ocasiones una clase se asocia consigo misma.



Agregación

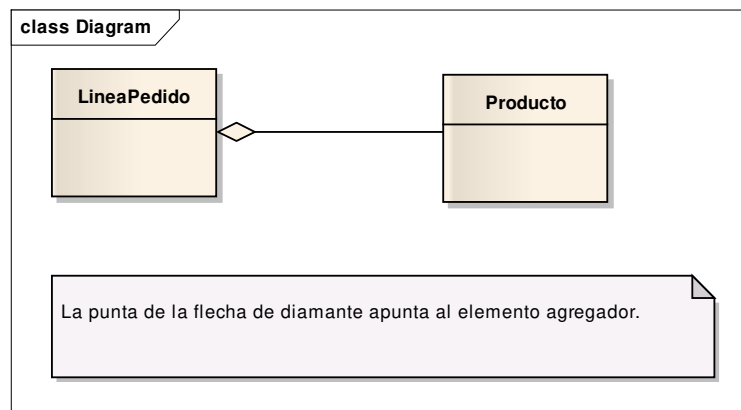
- **Agregación:**

- Es un tipo especial de asociación.
- Se produce **cuando una clase está formada por otras clases**. Se lee “**...es parte de...**” o “...está formado por...”
- El tiempo de vida del objeto incluido es **independiente** del que lo incluye.
- Se representa con una línea con punta de diamante hueca. La punta de diamante está en el lado del agregador.



Agregación

- Ejemplo: una línea de pedido está formada por productos. Si la línea de pedido se elimina, el producto sigue existiendo, no se elimina.



Composición

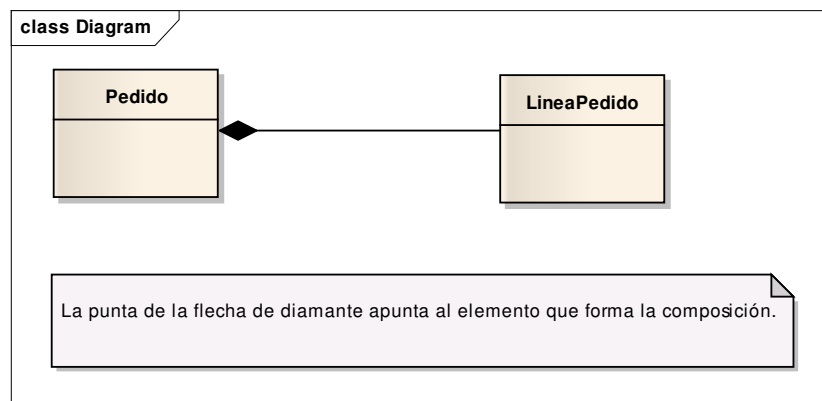
- **Composición:**

- Es un tipo especial de asociación (como agregación).
- Se produce **cuando una clase está formada por otras clases** (como agregación).
- El **tiempo de vida** del objeto incluido es **dependiente** del que lo **incluye** (distinto de agregación).
- Se representa con una línea con punta de diamante rellena.



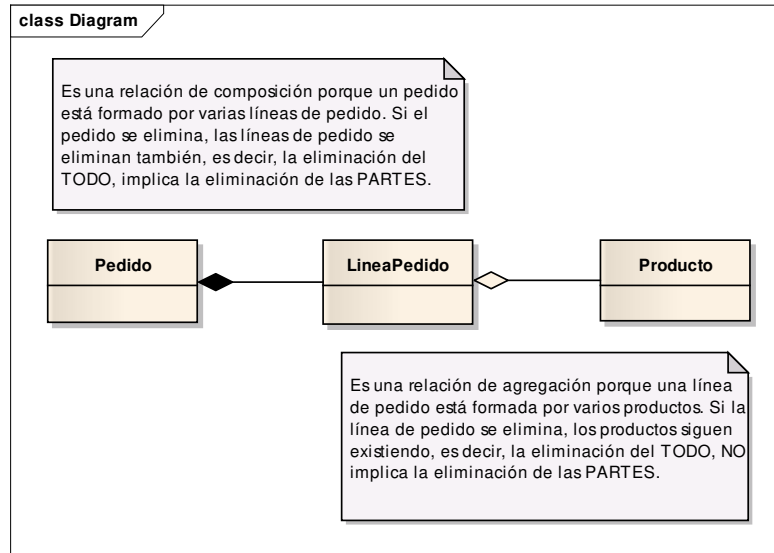
Composición

- Ejemplo: un pedido está formado por líneas de pedido. Si el pedido se elimina, se eliminarán las líneas asociadas al pedido.



Ejemplo de Agregación y Composición

- Ejemplo: un pedido está formado por líneas de pedido. A su vez, una línea de pedido está formada por productos.





Patrones Software

Tema 1-2:

Introducción

Introducción a los Patrones de Diseño

- Los analistas/diseñadores con gran experiencia aplican, de forma intuitiva y automática, criterios precisos que, en general, solucionan de forma elegante y efectiva los problemas de modelado software de sistemas reales.
- Usualmente estos diseñadores utilizan métodos, estructuras y subsistemas que son, a la vez, herramientas de diseño y partes de la solución final, de una manera que difícilmente puede transmitirse, en un sentido formal, a especialistas menos expertos.



Introducción a los Patrones de Diseño

- Los “ingenieros de software” se enfrentan cada día a multitud de problemas con distinto grado de dificultad.
- La “efectividad” de un “ingeniero” se mide por su rapidez y acierto en la diagnosis, identificación y resolución de tales problemas.
- El mejor “ingeniero” es el que más reutiliza la misma solución -matizada- para resolver problemas similares.



Introducción a los Patrones de Diseño

- Los diseñadores experimentados poseen un sentido especial que “detecta” la completitud, en un sentido eminentemente arquitectónico, de un determinado diseño, con independencia de las posibles métricas y paradigmas utilizados.
- Naturalmente lo ideal sería extraer la “esencia” de estos afortunados diseños para formular una “poción” que pudieran ingerir los diseñadores noveles.



Orígenes de los Patrones de Diseño

- Los trabajos de Christopher Alexander intentan identificar y resolver, en un marco descriptivo formal, problemas esenciales en el dominio de la arquitectura.
- El traslado de muchas de sus ideas al desarrollo de software ha parecido el más adecuado a los diseñadores.
- Alexander ha servido, en realidad, de catalizador de ciertas tendencias “constructivas” utilizadas en el diseño de sistemas software.



Orígenes de los Patrones de Diseño

- ¿Existe en verdad una parte común en los buenos diseños, a veces tan dispares entre sí? Christopher Alexander así lo afirma y da la siguiente definición de patrón:

“Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma”.



Orígenes de los Patrones de Diseño

- En 1987 Ward Cunningham y Kent Beck utilizan las ideas de Alexander para desarrollar un lenguaje de patrones como guía para los programadores de Smalltalk, dando lugar al libro "Using Pattern Languages for Object-Oriented Programs".
- En 1991 Jim Coplien publica el libro "Advanced C++ Programming Styles and Idioms", donde realiza un catalogo de "idioms" (especie de patrones).
- Entre 1990 y 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos como GoF (Gang of Four) "la banda de los cuatro") realizan el primer catálogo de patrones de diseño, que publican en el libro "**Design Patterns: Elements of Reusable Object-Oriented Software**".



Patrones de Software y OO

- Los **patrones** para el desarrollo de software son uno de los últimos avances de la Tecnología Orientada a Objetos.
- Los patrones se convierten en una parte muy importante en las Tecnologías Orientadas a Objetos para poder conseguir la **reutilización**. Entre los beneficios que se consiguen con la reutilización están:
 1. Reducción de tiempos.
 2. Disminución del esfuerzo de desarrollo y mantenimiento.
 3. Eficiencia.
 4. Consistencia.
 5. Fiabilidad.
- Las **metodologías** Orientadas a Objetos tienen como principio "*no reinventar la rueda*" para la resolución de diferentes problemas.



Patrones de Software + IS

- La **ingeniería del software** se enfrenta a problemas variados que hay que identificar para poder utilizar la misma solución (aunque matizada) con problemas similares.
- Los patrones son una forma documentada para resolver problemas de ingeniería del software.
- Se necesita algún esquema de documentación que permita la comunicación de resultados obtenidos por distintos ingenieros. No sirve solo con la documentación de líneas de código, hay que documentar análisis, diseños, arquitecturas, etc.
- El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.



Patrones de Software. Definición

- *Un patrón es una información que **captura la estructura esencial** y la **perspicacia** de una familia de **soluciones probadas con éxito** para un **problema repetitivo** que surge en un cierto contexto y sistema.*
- *Un patrón es una unidad de información nombrada, instructiva e intuitiva que **captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente** dentro de un cierto contexto.*
- *Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un conjunto de fuerzas que ocurren repetidamente en ese contexto y una cierta configuración software que permite a estas fuerzas resolverse por si mismas.*



Patrones de Software. Características

- **Solucionar un problema:** Los patrones capturan soluciones, no sólo principios o estrategias abstractas.
- **Ser un concepto probado:** Los patrones capturan soluciones demostradas, no teorías o especulaciones.
- **La solución no es obvia:** Muchas técnicas de solución de problemas tratan de hallar soluciones por medio de principios básicos. Los mejores patrones generan una solución a un problema de forma indirecta.
- **Describe participantes y relaciones entre ellos:** Los patrones no sólo describen módulos sino estructuras del sistema y mecanismos más complejos.



Patrones de Software. Clasificación

- **Patrones de arquitectura:** Expresa una organización o esquema estructural fundamental para sistemas software.
- **Patrones de diseño:** Proporciona un esquema para refinar los subsistemas o componentes de un sistema software, o las relaciones entre ellos.
- **Patrones de programación (Idioms patterns):** Un idioma es un patrón de bajo nivel de un lenguaje de programación específico.
- **Patrones de análisis:** Describen un conjunto de prácticas que aseguran la obtención de un buen modelo de un problema y su solución.
- **Patrones organizacionales:** Describen la estructura y prácticas de las organizaciones humanas, especialmente en las que producen, usan o administran software.



Patrones de Software. Clasificación

- También se puede hablar de otros tipos de patrones software, como pueden ser:
 1. Patrones de programación concurrente.
 2. Patrones de interfaz gráfica.
 3. Patrones de organización del código.
 4. Patrones de optimización de código.
 5. Patrones de robustez de código.
 6. Patrones de la fase de prueba.



Patrones de Software. Ventajas

- Ventajas de la utilización de patrones:
 1. Facilitan la comunicación interna.
 2. Ahorran tiempo y experimentos.
 3. Mejoran la calidad del diseño y la implementación.
 4. Son como “normas de productividad”.
 5. En Java facilitan su aprendizaje y comprender cómo está diseñado el propio lenguaje.



- ¿Patrones como moda?
 - Mejor como Intención.
- ¿Patrones como solución?
 - Mejor como núcleo de soluciones.
- ¿Patrones como normas?
 - Mejor como sugerencias.



Patrones Software

Tema 1-3:

Conceptos generales

Patrones de Diseño

- El objetivo de los patrones de diseño es guardar la experiencia en diseños de programas orientados a objetos (catálogos de patrones).
- Podemos encontrar patrones de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y reutilizables.
- Un diseñador que conoce algunos patrones puede aplicarlos inmediatamente a problemas de diseño sin tener que descubrirlos.
- Los patrones de diseño ayudan a un diseñador a conseguir un diseño correcto rápidamente.



Elementos de un patrón

- El **nombre del patrón** se utiliza para describir un problema de diseño, su solución, y consecuencias de forma resumida. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño.
- El **problema** describe cuando aplicar el patrón. Se explica el problema y su contexto. Podría describir problemas de diseño específicos tales como algoritmos o como objetos. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.



Elementos de un patrón

- La **solución** describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones.
La solución no describe un diseño particular o implementación, proveen una descripción abstracta de un problema de diseño y una disposición general de los elementos (clases y objetos en nuestro caso) que lo soluciona.
- Las **consecuencias** son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.



Elementos de un patrón

- Los patrones de diseño identifican las clases y objetos participantes, sus papeles, colaboraciones y comunicaciones y la distribución de responsabilidades.
- Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño común, que lo hace útil para la creación de diseños orientados a objetos reutilizables.
- Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.



Descripción de un patrón

- **Nombre:**
 - Un nombre descriptivo del patrón. El nombre es significativo y corto, fácil de recordar y asociar a la información que sigue. También se pueden incluir nombres alternativos, en caso de que los haya.
- **Propiedades:**
 - La clasificación del patrón. Tipo: creación, estructural, comportamiento. Nivel: Clase única, Componente (grupo de clases), Objeto.
- **Objetivo o Propósito:**
 - Breve explicación de las implicaciones del patrón.
- **Aplicabilidad:**
 - Cuando y porqué es deseable usar este patrón. En que situaciones es aplicable el patrón.



Descripción de un patrón

- **Estructura:**
 - Es el núcleo del patrón. Se describe una solución general al problema que el patrón soluciona. Esta descripción puede incluir, diagramas y texto que identifique la estructura del patrón, sus participantes y sus colaboraciones para mostrar como se soluciona el problema.
- **Consecuencias:**
 - Explica las implicaciones, buenas y malas, del uso de la solución.
- **Patrones relacionados:**
 - Otros patrones con los que está asociado o con los que tiene una relación estrecha.
- **Código de ejemplo:**
 - Ejemplo en código Java.



Cualidades de un patrón

- **Encapsulación y abstracción:**
 - Cada patrón **encapsula un problema bien definido** y su **solución** en un dominio particular y limitado.
 - Los patrones también sirven como abstracciones las cuales contienen dominios conocidos y experiencia.
- **Extensión y variabilidad:**
 - Cada patrón debería ser abierto por extensión o parametrización por otros patrones, de tal forma que pueden aplicarse juntos para solucionar un gran problema.
 - Un patrón solución debería ser también capaz de realizar una variedad infinita de implementaciones (de forma individual, y también en conjunción con otros patrones).



Cualidades de un patrón

- **Generatividad y composición:**
 - Cada patrón, una vez aplicado, genera un contexto resultante, el cual concuerda con el contexto inicial de uno o más de uno de los patrones del catálogo.
 - Esta subsecuencia de patrones y su composición con otros patrones podría ser aplicada progresivamente para conseguir la generación de un “todo” o solución completa.
- **Equilibrio:**
 - Cada patrón debe realizar algún tipo de balance entre sus efectos y restricciones.



Clasificación de los patrones

- **Patrones de creación:**
 - Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Éstas se suelen resolver dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades.
 - A menudo hay varios patrones de creación que se pueden aplicar en una misma situación. Otras veces se pueden combinar varios de ellos. En otros casos se debe elegir solo uno de ellos.
- **Patrones estructurales:**
 - Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.
- **Patrones de comportamiento:**
 - Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos. Nos permiten definir la comunicación entre los objetos de nuestro sistema y el flujo de la información entre los mismos.



Clasificación de los patrones

CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
Abstract Factory (O)	Adapter (C)	Chain of responsibility (O)
Builder (O)	Bridge (O)	Command (O)
Factory Method (C)	Composite (O)	Interpreter (C)
Prototype (O)	Decorator (O)	Iterator (O)
Singleton (O)	Facade (O)	Mediator (O)
	Flyweight (O)	Memento (O)
	Proxy (O)	Observer (O)
		State (O)
		Strategy (O)
		Template Method (C)
		Visitor (O)

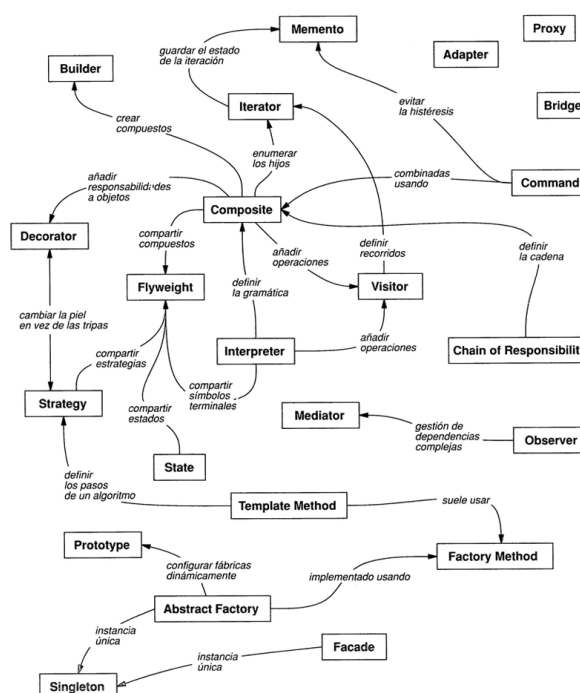
Ámbito: (C) Clase, (O) Objeto.

Clase: Relaciones entre clases y subclases. Herencia. Estáticas

Objeto: Relaciones entre objetos. Dinámicas.



Relación entre los patrones



Patrones y Frameworks

- **Framework:** Conjunto de clases que cooperan y forman un diseño reutilizable para un tipo específico de software. Un framework ofrece una **guía arquitectónica** partiendo el diseño en clases abstractas y definiendo sus responsabilidades y sus colaboraciones. Un desarrollador personaliza el marco de trabajo para una aplicación particular mediante herencia y composición de instancias de las clases del framework.
- Un framework no es generalmente una aplicación completa sino que a menudo le falta la **funcionalidad** de una aplicación específica.
- Una **aplicación**, en cambio, puede ser **construida** con **uno o más frameworks** insertando dicha funcionalidad.
- Un framework proporciona una guía arquitectónica para dividir el diseño en clases y definir sus responsabilidades y colaboraciones. Un framework dicta la arquitectura de la aplicación.



Patrones y Frameworks

- Los **frameworks** pueden incluir **patrones de diseño**.
- Los frameworks son ejecutables mientras que los patrones de diseño representan un conocimiento o experiencia sobre software.
- Un **framework** se puede ver como la **implementación de un sistema de patrones de diseño**.
- Los **frameworks** son de naturaleza **física**, mientras que los **patrones** son de naturaleza **lógica**: los frameworks son la realización física de uno o varios patrones de soluciones software; los patrones son las instrucciones de cómo implementar estas soluciones.



Patrones y Frameworks. Diferencias

- Los patrones de diseño son más abstractos que los frameworks:
 - *Los frameworks pueden ser codificados, pero solo los ejemplos de patrones de diseño pueden ser codificados. Los frameworks pueden ser escritos bajo lenguajes de programación y no solamente estudiados para ejecutarse mientras que los patrones de diseño tienen que ser implementados cada vez que son usados.*
- Los patrones de diseño son elementos arquitectónicos más pequeños que los Frameworks:
 - *Un framework contiene varios patrones de diseño, mientras que un patrón de diseño no contiene varios frameworks.*
- Los patrones de diseño están menos especializados que los frameworks:
 - *Los frameworks siempre tienen un dominio particular de aplicación, mientras que los patrones de diseño pueden ser utilizados en cualquier tipo de aplicación.*



Patrones Software

Tema 1-4:

Ejemplo

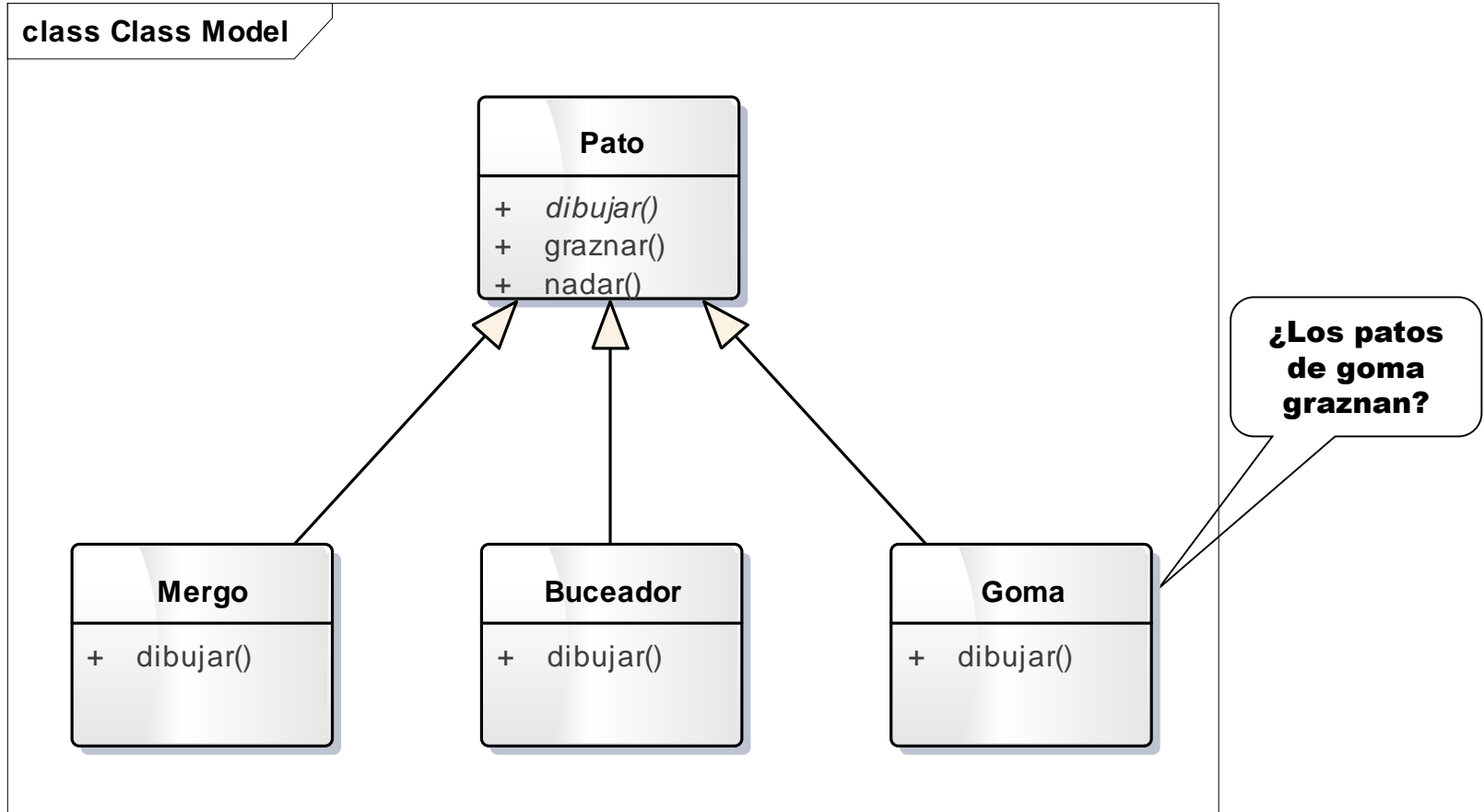
Introducción

Planteamiento del problema

- Se pretende hacer un juego que represente un simulador de patos.
- El juego permite visualizar una gran variedad de especies nadando en un lago y graznando.
- El diseño inicial del sistema usa técnicas estándar de POO y crea una jerarquía de clases.

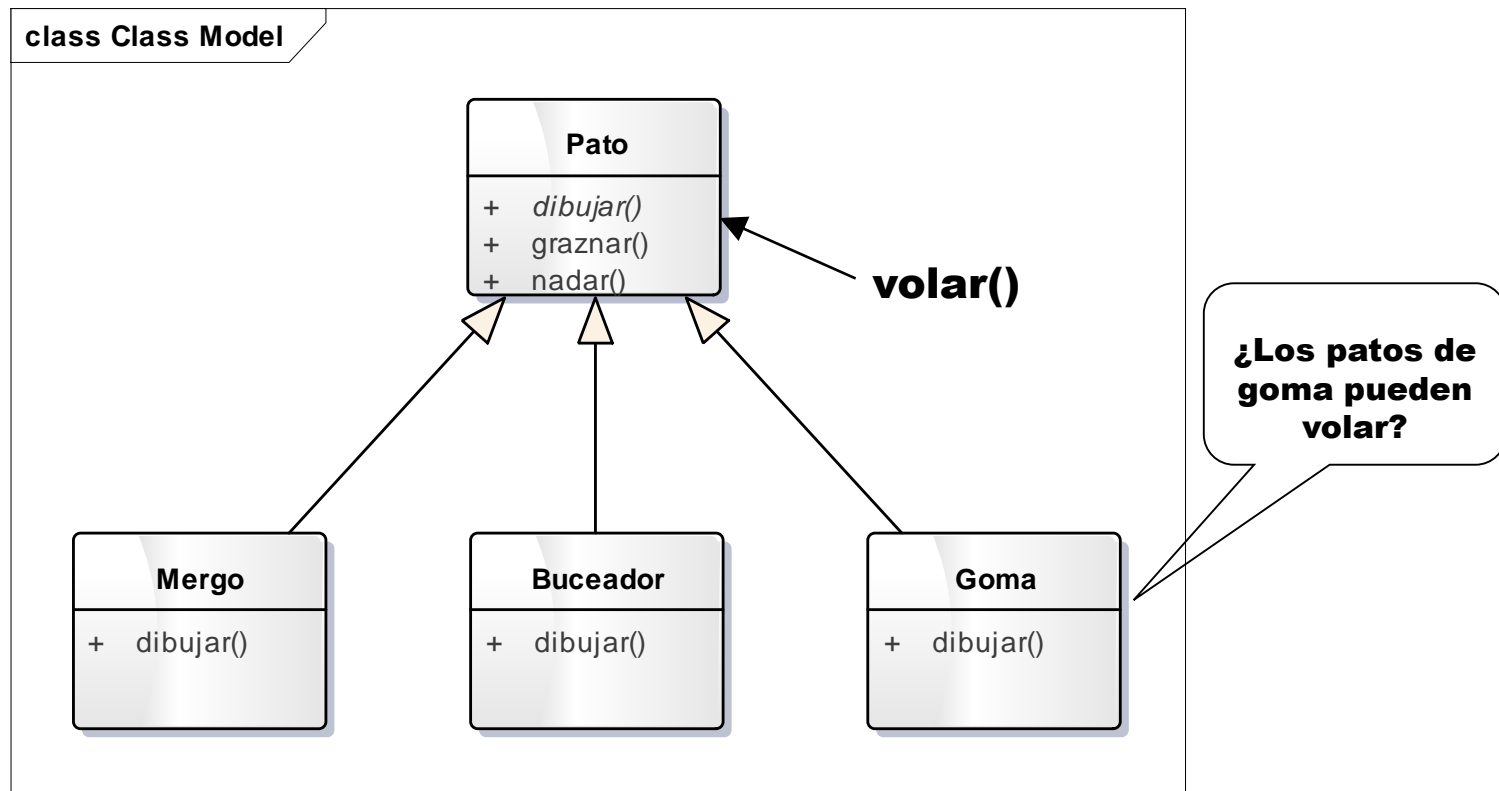


Primer análisis



Cambios en la especificación 1

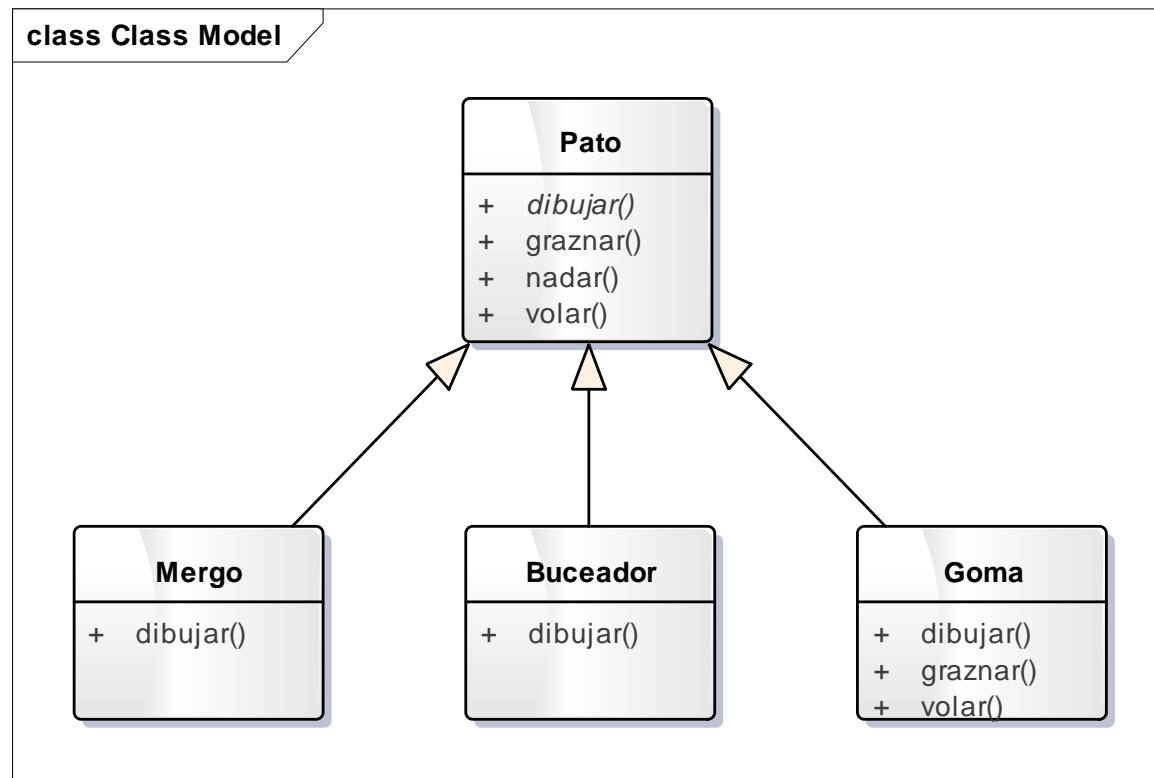
- Se desea añadir al simulador la posibilidad de que los patos vuelen.



Cambios en la especificación 1.

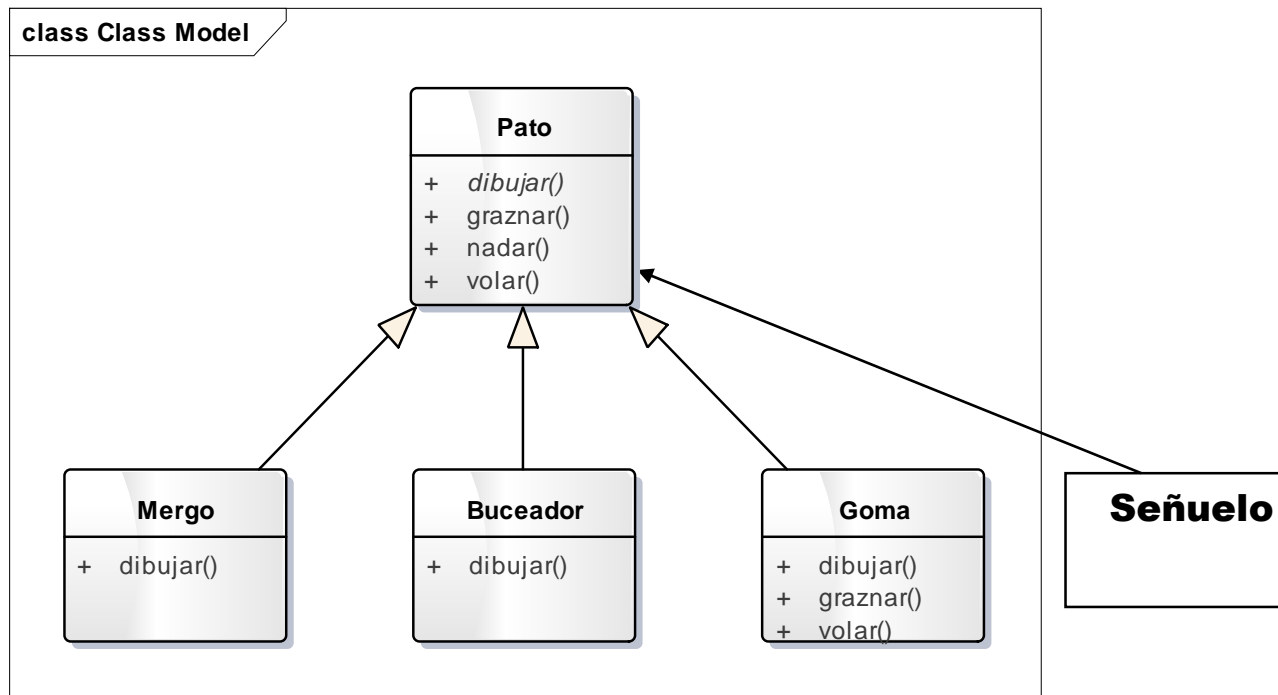
Solución

- La clase Pato de Goma redefine los métodos graznar() y volar() dejándolos vacíos.



Cambios en la especificación 2

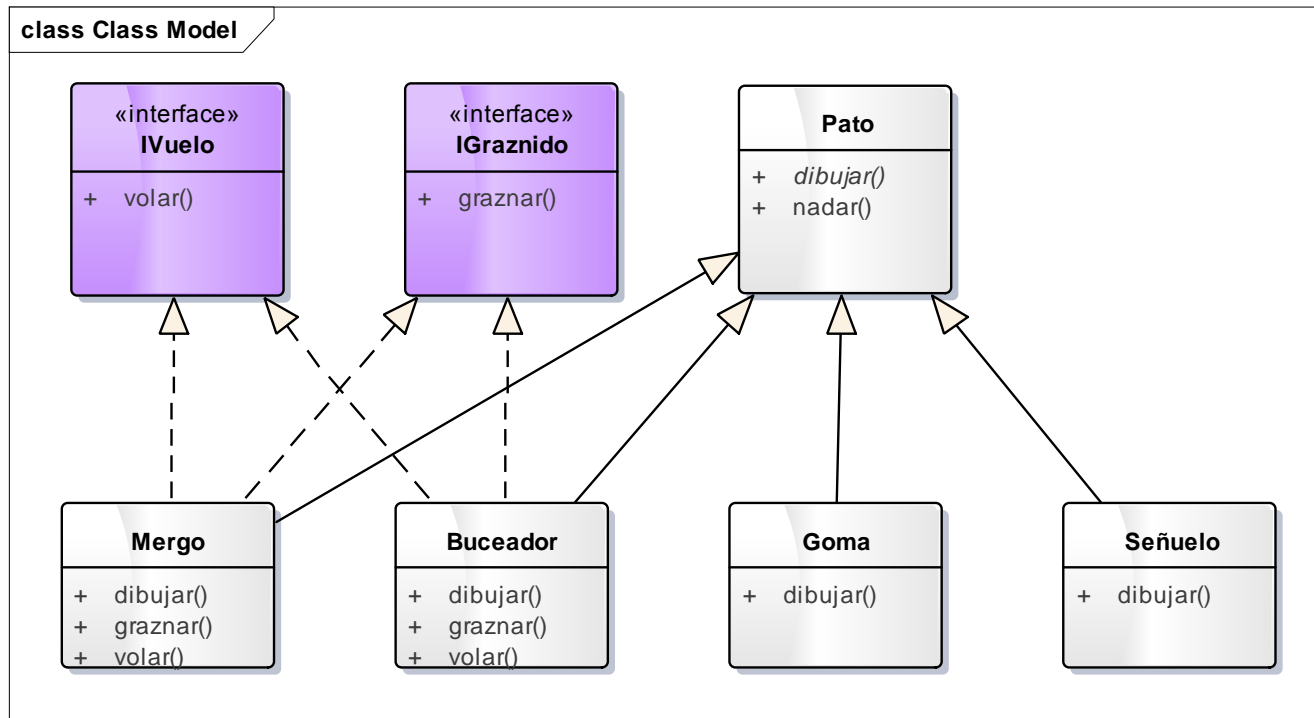
- Pero ¿que pasa si añadimos la clase Pato de Señuelo? ¿Redefinimos los métodos graznar() y volar() dejándolos vacíos?.



Cambios en la especificación 2.

Solución

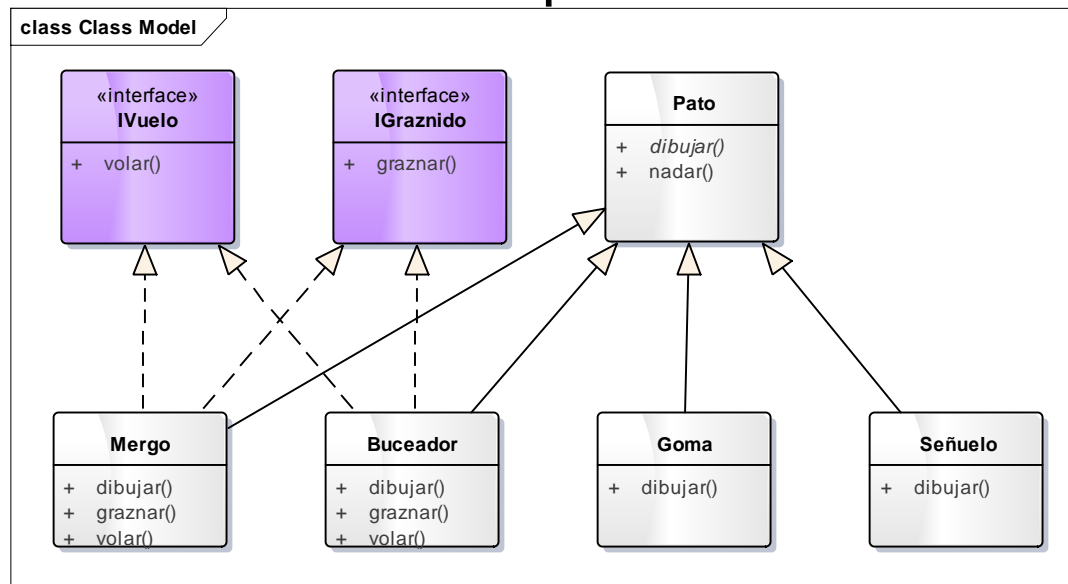
- Mejor creamos interfaces con los métodos volar y graznar y aquellas clases que los necesiten los implementan.



Cambios en la especificación 2.

Solución

- ¿Pero que pasa si tenemos un montón de clases de patos que necesitan volar o graznar? Todos deben implementar los métodos. No reutilizamos el código de estos comportamientos. Por lo tanto el mantenimiento se hace imposible.



Principio de Diseño

- La herencia no ha resuelto los problemas de cambio de comportamiento en las subclases, ya que no todas las subclases tienen el mismo comportamiento.
- Las interfaces están bien en principio, ya que solo las clases que necesitan un determinado comportamiento las implementan. Pero cuando tenemos que modificar un comportamiento debemos ir al código de todas las clases que han implementado el interface.
- En Java los métodos de las interfaces no tienen código por lo tanto no hay reutilización.



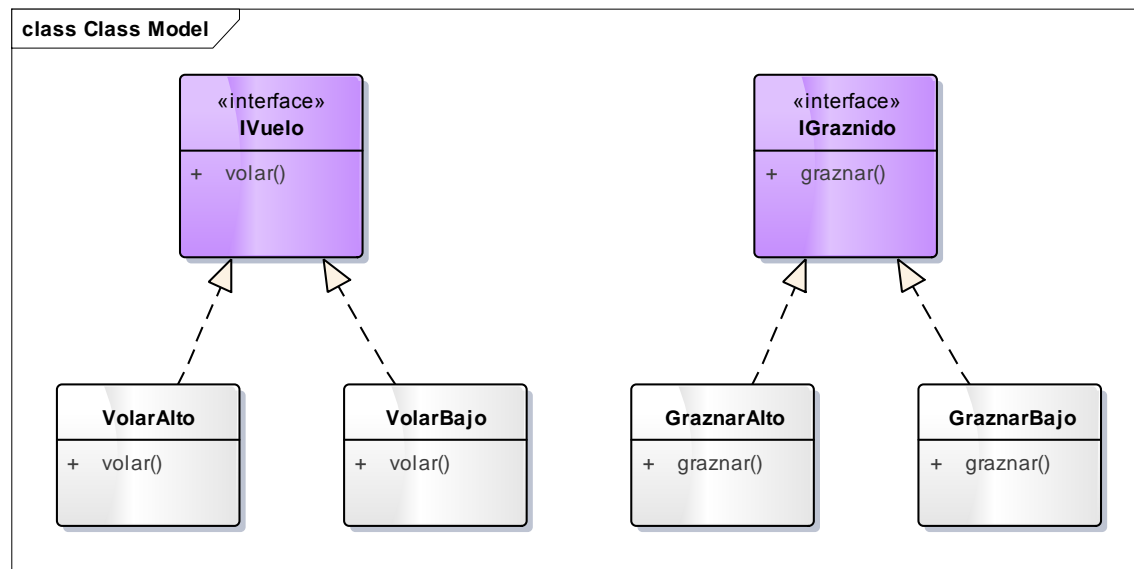
Principio de Diseño

- Principio de diseño:
 - Identifica aquellos aspectos de tu aplicación que varían y sepáralos de aquellos que no varían.
- Es decir, si tienes código que cambia con cada nuevo requerimiento, debes separarlo del resto que permanece invariable.
- Otra forma de verlo: Si separas las partes que varían y las encapsulas, más tarde podrás cambiarlas sin que afecten a otras partes.
- Los patrones precisamente proporcionan formas de mantener independientes ciertas partes del sistema.



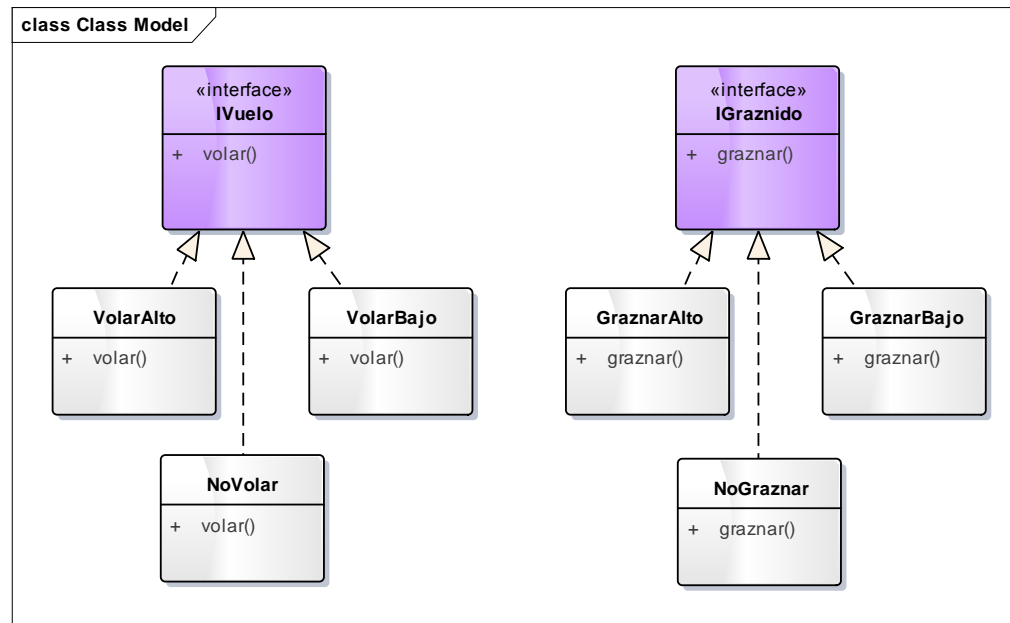
Separando comportamientos

- Para separar los comportamientos vamos a crear dos conjuntos de clases uno para volar y otro para graznar.
- Queremos tener un comportamiento predefinido de volar y graznar pero que incluso podamos cambiar en tiempo de ejecución.



Separando comportamientos

- Con este diseño otros tipos de objetos pueden reutilizar los comportamientos de volar y graznar ya que no están escondidos en la clase Pato.
- Podemos añadir nuevos comportamientos sin modificar ningún comportamiento anterior ni las clases que heredan de Pato.



Integrando comportamientos

- Ahora la clase Pato delega su comportamiento, en vez de definirlo dentro.
- Estamos utilizando el patrón de comportamiento **estrategia** que define una familia de algoritmos, los encapsula y los hace intercambiables, estos pueden cambiar de forma independiente al uso que hace el cliente de ellos.



Estructura final

class Class Model

