El método de ingeniería

Consideraciones generales sobre el posicionamiento de la informática.

La incorporación de procedimientos informáticos a una organización requiere como paso previo la racionalización de los procesos de dicha organización. Para ello, además de visualizar como es lógico la organización según fines y posiciones (organigrama), se estructurarán los procesos de negocio según grafos de procedimientos (que vincularán actividades) y podrán emplearse matrices que relacionen estas funciones o actividades con las distintas posiciones y con las herramientas que se necesita para llevarlas a cabo.

El objetivo último de todo esto será conseguir promotividad en los sistemas: es decir, la incorporación de inteligencia contextual a los sistemas informáticos para que sean capaces de analizar el trabajo a llevar a cabo en la organización de forma que, en el momento oportuno, en el lugar oportuno y a la posición oportuna, y de forma coordinada con las demás posiciones de la organización, se le proporcionen automáticamente las herramientas adecuadas para llevarlo a cabo.

Enfoque de la Ingeniería Informática

Ámbitos de actuación del ingeniero informático

- Programa de desarrollos.
- Planificación de proyectos.
- Adecuación de tecnología.
- Estudios económicos.

Estructura general del método de ingeniería

Un método de ingeniería ha de desarrollar los siguientes aspectos, en este orden:

1. *Memoria justificativa*: será, a fin de cuentas, la "venta" del proyecto a los directivos que al final deberán aprobarlo. Debe describir, de manera clara y concisa, las ventajas que prestará el sistema a los ámbitos en donde se va a instalar (el sistema nunca habrá de justificarse por los atributos de su tecnología).

    Introducción o enfoque:

Captar la atención.
Ventajas del planteamiento.
Atributos de la solución (después de las ventajas).
Referencias (positivas y negativas).
FEED BACK.

Determinación de la situación actual.

Prueba de hipótesis: determina qué es lo que deberá soportar el sistema.
Análisis DAFO (oportunidades).

Alcance y objetivos.

Exposición general de las pretensiones del sistema en un lenguaje claro y sencillo.

2. *Modelo acoplado y eficiente*. Una vez presentada la memoria del proyecto, se procederá con el modelado. Los mejores modelos son los que se integran (de ahí lo de "acoplado") con las necesidades de la organización.

3. *Plataforma tecnológica (despliegue)*. Se trata de considerar el despliegue que habrá de sustentar el sistema.

4. *Planificación de actividades y recursos*. Deberá predecirse la historia del proyecto. Para la planificación pueden emplearse métodos basados en grafos, como PERT o Roy (ver siguiente tema).

5. *Estudios económicos*. El sistema puesto en el tiempo y costeado (Cash flows, etc.).

Clasificación del software empresarial.

Una posible clasificación de la solución informática es: *commodity* (software que es subcontratado a un tercero –*outsourcing*–), *facility*, *pragmatic*, *visionary* (software en busca de nuevas líneas de negocio – en muchos casos prototipos).

Arquitecturas de Integración de Sistemas.

El nuevo sistema habrá de integrarse en el despliegue actual de la organización, de forma que se respete el legado o *legacy* que, a fin de

cuentas, representa el "saber hacer" de la organización que le ha permitido llegar a su situación actual.

Hay varios enfoques:

- Basadas en flujos de datos: también conocidas como ETL (Extracción, Transformación y Carga (*Load)*).
- Virtualización de entornos.

Planificación.

PERT.

Conceptos.

Una red PERT es matemáticamente un grafo (V, A), donde V son una serie de vértices llamados sucesos (por convención identificados numéricamente) y $A \subseteq V \times V$ serán las actividades.
Actividad: si se realiza, no se puede interferir. Cada actividad tiene asociada una duración. Han de ser matemáticamente distinguibles; es decir, (3, 4) representa una actividad y no podrá por tanto haber más de una "flecha" del nodo 3 al nodo 4.
Para solucionar el anterior problema y otros que se mencionarán se recurre a actividades virtuales (duración 0).

Dependencias no deseadas.

Escalera de numeración.

Sirve para comprobar que el orden establecido *a priori* entre las actividades es el correcto – i.e., que las dependencias van siempre *hacia delante* en el grafo.

Cálculo de tiempos. Red ejemplo.

Tiempo mínimo de suceso, *tiempo máximo de suceso* y *holgura de suceso*.
Se dice que una red es hipercrítica cuando la duración mínima calculada del proyecto es menor a la duración impuesta (con lo cual saldrán holguras negativas). Será crítica cuando es igual.

TAPC(i, j) = Tmín(i).
TTPC(i, j) = TTPT(i, j) - d(i, j).

TTPT(i, j) = Tmáx(j).
TAPT(i, j) = TAPC + d(i, j).

El *camino crítico* será aquél que una sucesos con holgura cero. Puede haber más de un camino crítico, pero es claro que todos ellos tendrán la misma duración – si uno de ellos tuviera una duración mayor, el otro tendría sucesos con holgura.

La *holgura de actividad* se calcula en función de la holgura de suceso:

- Holgura total: HTOTAL(i, j) = Tmáx(j) - Tmín(i) - d(i, j) = TTPT(i, j) - TAPC(i, j) - d(i, j).
- Holgura libre: HLIBRE(i, j) = Tmín(j) - Tmín(i) - d(i, j). Una actividad puede consumir su holgura libre sin robársela a las actividades que dependen de ella.
- Holgura independiente: HINDEP(i, j) = Tmín(j) - Tmáx(i) - d(i, j). Es la holgura que se puede consumir con absoluta independencia (no roba holgura a otras actividades). Puede ser negativa si hay varias actividades encadenadas y otro camino de mayor duración – el incremento de una de las actividades encadenadas sin robar holgura a la anterior robará holgura a la siguiente. [Revisar]

Las actividades del camino crítico tendrán holgura nula.

Actividades virtuales de avance y retardo.

Una *actividad de avance* es una actividad virtual a la que se asigna tiempo entre dos actividades A y B para consumir holgura ("ociosidad") de la actividad A y redistribuir los recursos (por ejemplo, que el equipo de la actividad A pase a apoyar a otra cuando termine – "recursos compatibles"). *Actividad de retardo* es ….

Aceleración de la red a coste mínimo.

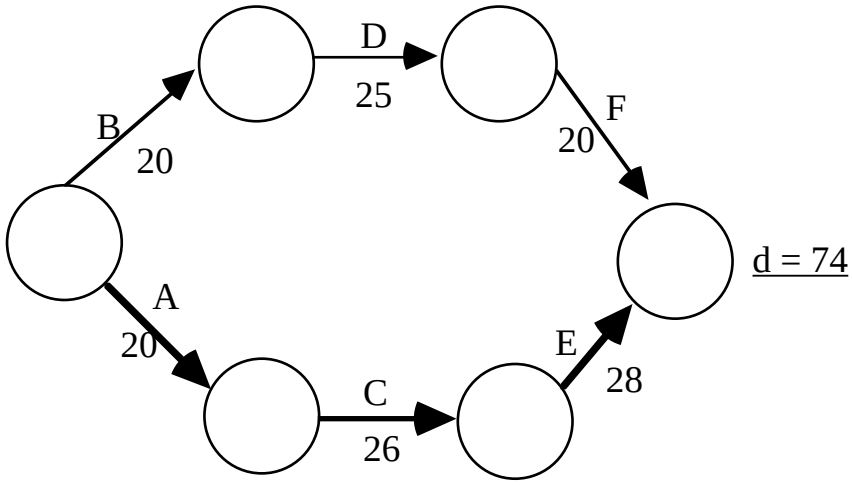
Se parte de que una actividad tiene asociadas *costes directos* y *costes indirectos*. Los costes directos es lógico que disminuyan cuanto más se haga durar la actividad, pero los indirectos podrán aumentar cuanto más dure (p.ej. gastos de alquiler de un almacén).
La aceleración se efectúa con un sencillo algoritmo que linealiza la función de coste directo y no tiene en cuenta los costes indirectos.
La conclusión es que si los costes indirectos son significativos, no se puede optimizar sólo con los directos, si no que habrá que tener en cuenta también

los indirectos y hallar el mínimo (lo que se hará mediante optimización matemática de la función).

Ejemplo de aceleración a coste mínimo. Tabla de datos:

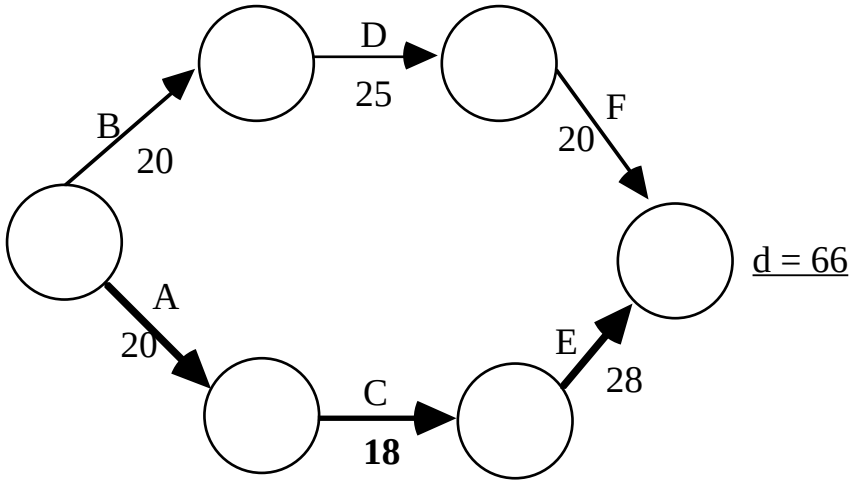
Actividad	Dur. mínima	Dur. máxima	Coste unitario
A	12	20	6,25
B	15	20	10
C	18	26	5
D	20	25	8
E	25	28	10
F	18	20	25

Red inicial:



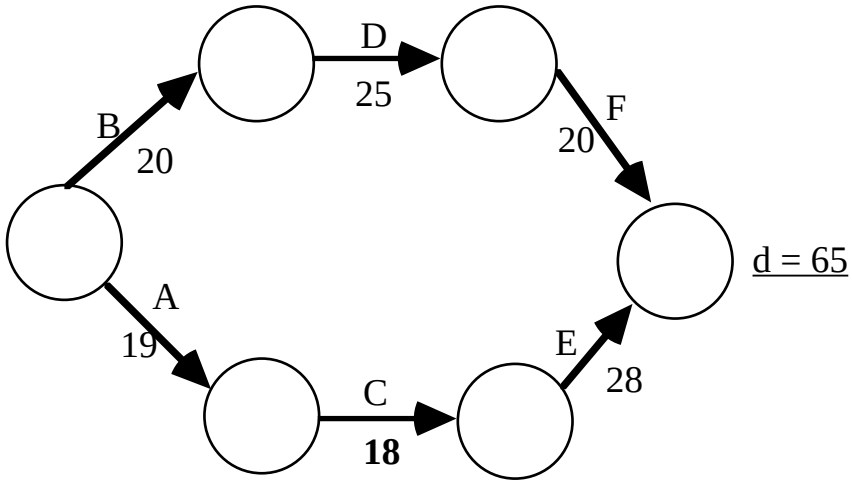
Iteración 1:

El mínimo es, evidentemente, reducir C hasta el mínimo a coste 5 por unidad.



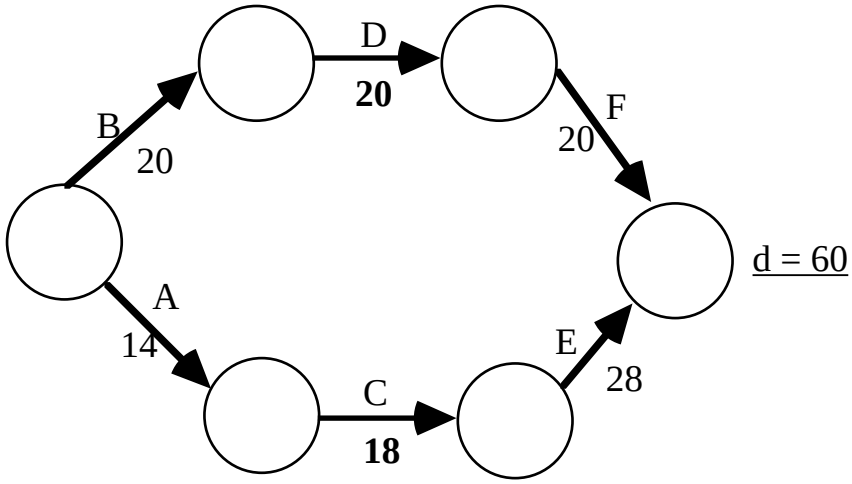
Iteración 2:

El mínimo es reducir A a coste 6,25 por unidad, pero sólo se puede reducir una unidad de momento, ya que las actividades B, D y F pasan también a ser críticas (ya que suman 65).



Iteración 3:

Ahora es necesario reducir los dos caminos a la vez. El mínimo es reducir A, D 5 unidades a coste 6,25 + 8 = 14,25 por unidad.



Y así sucesivamente. Iteración 4: Se reducen A, B 2 unidades; iteración 5: se reducen B, E 3 unidades. La duración mínima del ejemplo es de 55.

No obstante, se deberá tener en cuenta la gráfica de costes, dibujando el coste frente a la duración del proyecto. Debido a que en las optimizaciones finales hay que reducir varias actividades a la vez, es frecuente que el coste

se dispare y que no interese reducir al máximo sino quedarse en un estadío intermedio.

Optimización sin aceleraciones.

Consiste en bajar el coste de la red manteniendo la duración del proyecto. Es decir, alargar actividades con holgura (libre o independiente) bajando costes, reduciendo primero las más caras.

Una vez hecha esta optimización, hay que revisar las nuevas holguras, ya que al disminuirlas se pierde capacidad de gestión – como la reasignación de recursos comentada en el apartado *Actividades de avance y retardo*.

Optimización con aceleraciones.

Es una *aceleración de la red a coste mínimo* en la que se tiene en cuenta la *optimización sin aceleraciones*. Es una aceleración a coste mínimo en la que, en cada iteración del algoritmo, hay que considerar también la posibilidad de reducir más un camino crítico que otro, generando una holgura en éste que puede ser utilizada para aplicar la optimización sin aceleraciones. Si resulta un menor coste por unidad de reducción, ésta será la alternativa a escoger.

Debido a que el algoritmo es más complejo, es más conveniente aplicar el método de resolución SIMPLEX.

El siguiente problema de programación lineal se plantea para una duración de la red λ dada.

T(i,j)	Duración de la actividad.
c(i,j)	Coste por unidad de tiempo. Puesto que el coste es por reducción, se tendrá que $c(i,j) \leq 0$.

Variables:

T(i,j)

Función objetivo:

$$\min \sum_{(i,j)\in A} c(i,j) \cdot T(i,j)$$

Restricciones:

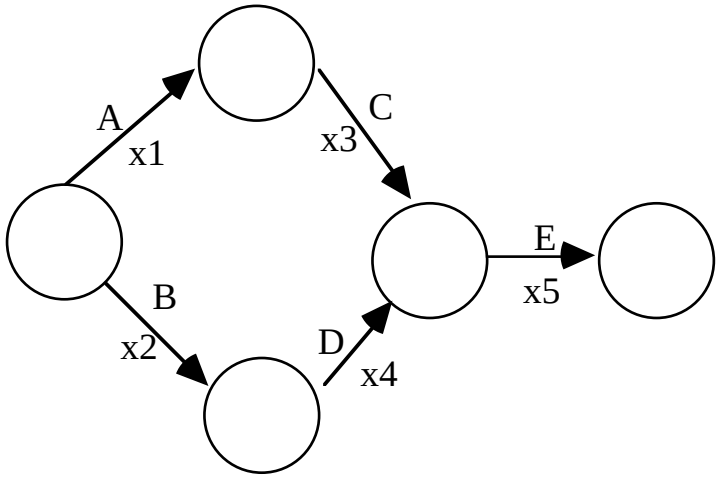
$T(i,j) \leq D_N(i,j)$ {Duración normal}

$T(i,j) \geq D_T(i,j)$ {Duración tope}

$\sum_{(i,j)\in K_n} T(i,j) \leq \lambda$ {Todos los caminos de duración menor que λ}

Es necesario ir viendo la topología de la red, ya que puede ser preciso detener la optimización en un punto intermedio si la solución óptima no es factible – por ejemplo podría superarse la nivelación de gasto para ese mes, como en el caso anterior la pérdida de holguras implica pérdida de capacidad de gestión, etc.

Red ejemplo:



Actividad	Dur. normal	Dur. tope	Coste unitario reducc.
A	3	1	2
B	5	1	7
C	11	4	2
D	8	2	8
E	6	3	6

$\min Z = -2x_1 - 7x_2 - 2x_3 - 8x_4 - 6x_5$

$\max (-Z) = 2x_1 + 7x_2 + 2x_3 + 8x_4 + 6x_5$

$x_1 \leq 3 \quad x_2 \leq 5 \quad x_3 \leq 11 \quad x_4 \leq 8 \quad x_5 \leq 6$

$x_1 \geq 1 \quad x_2 \geq 1 \quad x_3 \geq 4 \quad x_4 \geq 2 \quad x_5 \geq 3$

$x_1 + x_3 + x_5 \leq \lambda$

$x_2 + x_4 + x_5 \leq \lambda$

Orientación a objetos.

Análisis y diseño.

La verdadera aportación de la orientación a objetos (en adelante O. O.) es en el campo de análisis y diseño, aunque surgiera en el contexto de la programación. La aplicación del paradigma a este campo se centra en modelos organizados alrededor de conceptos del mundo real. Estos modelos se realizan en base al concepto de **objeto**.
Un **objeto** combina *estructura de datos* y *comportamientos* en una única entidad.
Las aplicaciones de los modelos O.O. son, entre otras, la comprensión del dominio de la aplicación, la comunicación con expertos, la modelización de organizaciones, la preparación de documentación y, por supuesto, el diseño de programas y bases de datos.

Características.

En general se coincide en que la orientación a objetos se fundamenta en los siguientes aspectos o pilares:

- Identidad
- Clasificación
- Polimorfismo
- Herencia

aunque el alcance y orientación de los mismos es muy discutido.

Identidad.

La identidad de un objeto es independiente del contenido de sus atributos. Puesto que los objetos intentan modelar el mundo real, la lógica para otorgar identidad a un objeto puede ser muy compleja —un objeto simplemente "existe"—, aunque en los entornos de programación se simplifica a un "manejador" o "puntero" que se utiliza para referenciarlo unívocamente (y que se otorga en el momento en que se crea en una clase, algo que será discutido más tarde en el apartado de *clasificación*).

Así pues, no existe el concepto de *llave* de las bases de datos.

Una posible clasificación de los objetos bajo el mundo real es en **concretos** y **conceptuales** (lo que no se debe confundir con ***clases abstractas***).

Clasificación.
Un objeto puede estar clasificado en ninguna, una o varias clases y puede tener sus propios atributos y métodos además de los propios de la clase.

La **clasificación** se realiza observando los objetos del mundo real y las características relevantes en el contexto que puedan compartir una serie de objetos. Es, por tanto, arbitraria.

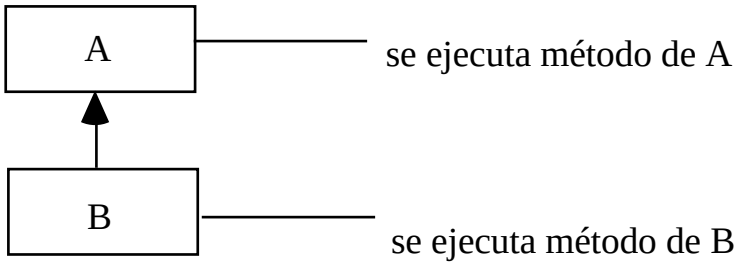
Polimorfismo.
Una definición genérica de polimorfismo es que un mismo *método,* con un alcance semántico similar, puede tener distintas implementaciones (valores) en distintos objetos.

En la práctica, este concepto entra también en discusión, y muchos entornos de programación acotan su alcance bajo el paraguas de la *herencia*, que se discutirá a continuación. Además, el concepto de "alcance semántico similar" es frecuentemente reducido a "misma firma", donde *firma* es un concepto que depende del lenguaje de programación y que consiste básicamente en "los mismos argumentos (tipos de dato) de entrada y salida".

Se ha hablado de *método* sin discutir este concepto. Un método es la implementación de una **operación**; es decir, de un **comportamiento** en el mundo real.

Herencia.
Consiste en reducir —*factorizar*— propiedades comunes a una serie de clases a una "superclase", de la que *heredarán*. El concepto de herencia es discutible y puede ser problemático. Frecuentemente se combina el polimorfismo con la herencia.

Idealmente, cada objeto ejecuta el método de una operación desde el punto de vista de la clase y relaciones con que esté interactuando. En el siguiente ejemplo, A y B implementan una operación polimórfica con sendos métodos (distintos).



Metodología OMT.

Fue propuesta por J. Rumbaugh. Tres fases: análisis, diseño e implementación.

Introducción.

El énfasis en la programación orientada a objetos (POO) puede ser considerado un paso atrás en la ingeniería del software, dado que hace excesivo hincapié en los mecanismos y artefactos de la implementación, creando una cierta "cerrazón mental". Los beneficios reales de la OO están en el campo del análisis y diseño. Desde este punto de vista, un planteamiento OO obliga a pensar y trabajar en términos del dominio de la aplicación.. Así pues, la identificación y organización de conceptos del dominio de la aplicación es la esencia de un desarrollo orientado a objetos. Algunos autores opinan que, precisamente, esta esencia es la parte más *dura* del desarrollo de software (en contraposición a los "accidentes" de su implementación en un lenguaje o tecnología concreta).

El desarrollo O.O. es un proceso independiente del lenguaje de programación hasta las últimas etapas. NO ES UNA TÉCNICA DE PROGRAMACIÓN sino de modelado, especificación, análisis…

Fases.

De forma general, durante el análisis se construye un modelo del dominio de la aplicación el cual se concretará más tarde con los artefactos de implementación.

1. Análisis. **Objetos del dominio de la aplicación**. Definiciones iniciales. Expresa lo que se desea de la aplicación (casos de uso). No entra en detalles de implementación.

2. Diseño del sistema.	**Posibilidades de la tecnología**. Arquitecturas de todo tipo que soportan las necesidades del sistema. Subsistemas. Estrategias para abordar el problema. Asignación de recursos.
3. Diseño técnico.	**Artefactos de implementación**. Estructuras de datos + algoritmos.
4. Implementación.	No tiene necesariamente que ser en un "lenguaje orientado a objetos" (POO).

Detrás de los objetos del dominio de la aplicación (definidos en Análisis) habrá una base de datos.

Consideraciones.

- Con respecto a los **objetos**: sólo se añade lo necesario para representar el modelo en cuestión (*economía del modelado*).
- No existe, evidentemente, un análisis único para un problema.

Modelos de objetos.

Para evitar el uso poco claro del término *objeto*, se habla de instancias de objetos y clases de objetos. Una *clase* tiene un mismo esquema de atributos y relaciones con otros objetos. Los objetos de una clase han de compartir un propósito semántico común.

Pero ¿cuál es la razón principal de agrupar los objetos en clases? La respuesta es que las clases son fundamentales para modelar, ya que permiten la abstracción y factorización del problema – en contraposición a pensar en una amalgama de objetos concretos del mundo real.

Los **diagramas de objetos** pueden ser *diagramas de clases* y *diagramas de instancias*. En ellos, pueden omitirse los atributos y sus descriptores si no son relevantes para entender el modelo [en UML se añade el concepto de "niveles semánticos"].

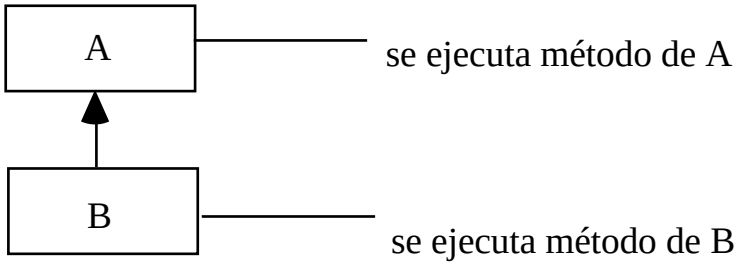
Atributos

Los atributos no son objetos – no tienen identidad. Así pues, dos atributos numéricos de valor 17 son indistinguibles.

Operaciones y métodos.

Una *operación* modela un **comportamiento** y tiene siempre un objeto destino –muchas veces implementado como un argumento implícito (*this*)–. Un *método* es una implementación de una operación. Puede haber más de un método para una misma operación si se da el polimorfismo. En este caso, debe de haber implementaciones conceptualmente válidias del método en

todos los niveles de la jerarquía de herencia, y se ejecutará la que corresponda según el nivel en que se proyecte el objeto.



Las operaciones pueden y deben ser omitidas en los diagramas de alto nivel.
Se llaman *operaciones **query*** las que no modifican el estado del objeto.

Links y asociaciones.

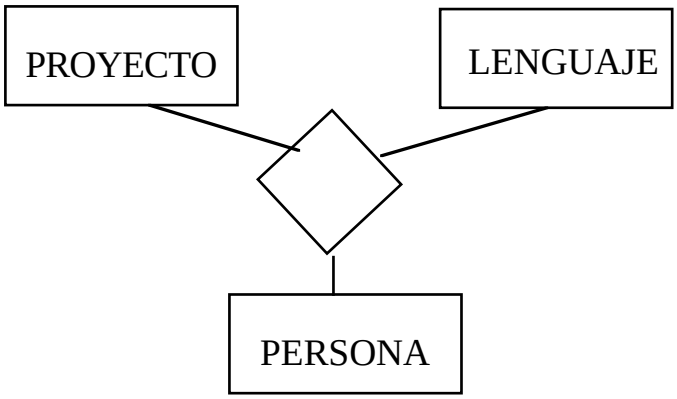
Las *asociaciones* son vínculos para transportar mensajes. Pueden ser binarias o de orden superior, que pueden —y en muchos casos es conveniente— reducirse a asociaciones binarias.
En OMT, son bidireccionales, algo que se corrige en UML, donde se permite especificar su *navegabilidad*.

La multiplicidad de las asociaciones binarias es fácil de representar:

- ○ Cero o una (opción).
- ● Cero o más.

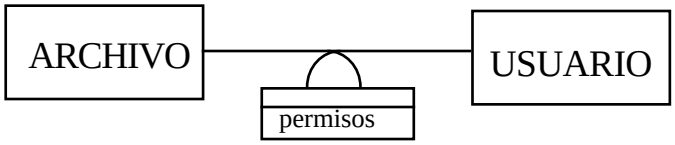
1 a 1

La **multiplicidad** es algo que se tiende a subestimar cuando se modela una aplicación, algo que limitará sus posibilidades. En cualquier caso, es claro que depende del dominio en cuestión. Por ejemplo, en una aplicación para administrar la contabilidad de una empresa, será irrelevante considerar que un empleado puede trabajar en varias. Pero esto no será así en una aplicación de Hacienda.
El problema de las asociaciones ternarias es **entender su multiplicidad**. No se puede utilizar la anterior notación porque resultaría ambigua.



(Ejemplo de asociación ternaria).

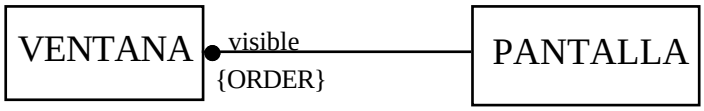
Los *link* o vínculos entre dos instancias no son "punteros", sino que son entidades que también son susceptibles de tener atributos independientes de los objetos. En realidad, un link es un objeto (y una asociación será pues una clase; clase asociación). En OMT, se dibuja esta clase "colgándola" de la asociación. En el caso especial de una asociación 1-a-N ó N-a-1, sería factible conceptualmente incorporar los atributos y asociaciones del objeto asociación en el término de multiplicidad 1.



En este ejemplo, la propiedad *permisos* es algo que depende del vínculo ARCHIVO-USUARIO.

No se debe confundir la presencia de una clase asociación con una asociación ternaria. Una instancia de una clase asociación existe si y sólo si hay un link entre los dos objetos, y ésta es única. Sin embargo, en una asociación ternaria, el tercer asociando puede tener multiplicidad varios. Así, en el anterior ejemplo, no puede existir más de un atributo "permisos" para unalink entre dos objetos ARCHIVO-USUARIO.

Usualmente los objetos en el/los términos "varios" de una asociación no tienen un orden explícito (son, matemáticamente, elementos de un conjunto –producto cartesiano–). Pero si semánticamente existe un orden, se puede añadir esta información al modelo mediante una anotación {ORDEN}.

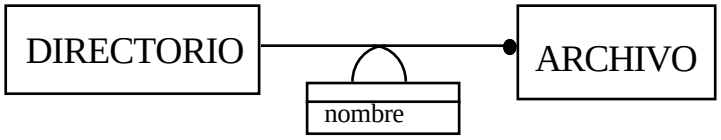


Calificación.

Consiste en incorporar un selector que permite "seleccionar" un subconjunto del término varios de una asociación. Es una forma que sustituye a la asociación ternaria, o bien a la clase asociación (dependiendo de la multiplicidad), pero que añade semántica al modelo (y se evita introducir la asociación ternaria si fuera el caso); además de añadir navegabilidad o direccionalidad, puesto que indica un sentido lógico en el que se atraviesa la asociación para llegar al otro término.

El ejemplo más sencillo es el siguiente:

(Ejemplo de selector)

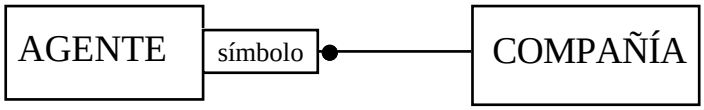


(Versión sin calificación, con distinta semántica)

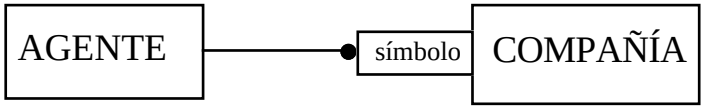
En este caso, la multiplicidad 1-a-varios se reduce a una multiplicidad 1-a-1 en presencia del selector. Aquí se aprecia las ventajas semánticas de la calificación. El primer modelo está expresando que un nombre de archivo determina un único archivo, algo que no se expresa en el segundo.

La calificación es especialmente útil cuando el selector no tiene la suficiente importancia, o bien resulta difícil otorgarle una identidad, como para ser considerado un objeto; como es el caso del anterior ejemplo.

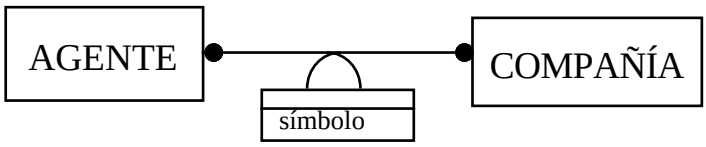
Además de esta calificación simple podemos considerar una **calificación avanzada** (que desaparece en UML). Ejemplo: cada agente de bolsa identifica las acciones de cada compañía con un símbolo.



Opción 1. Un agente, a través de un símbolo, selecciona una compañía.



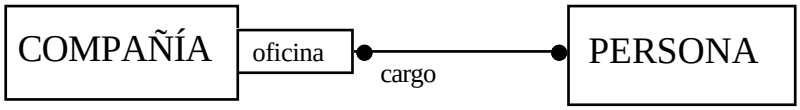
Opción 2. Una compañía, a través de un símbolo, selecciona varios agentes.



Opción 3. Sin calificación. Se pierde la semántica de que el símbolo determina una única compañía (opción 1) o un único agente (opción 2).

En este caso, la presencia de multiplicidad varios en ambos términos hace viables las dos opciones (1 y 2).

En los ejemplos anteriores el calificador siempre seleccionaba una única instancia del término calificado, pero esto no ocurre siempre: sea el caso de una compañía en cuyas oficinas tiene varias personas trabajando –cada una con su correspondiente cargo– y una persona puede tener cargos en varias compañías.



Modelo con calificación.

(ABC, Principal, presidente, Rogelio Lisa)
(ABC, Principal, tesorero, Juan Izquierdo)
(ABC, Principal, director, José Díaz)
(ABC, Principal, director, Juana Díaz)
(ABC, Principal, director, Marta Rojo)
(XYZ, Madrid, presidente, Marta Rojo)

Ejemplo de tuplas para la asociación.

Agregación.

Es una asociación con una semántica más rica. Intuivitamente, trata de capturar una relación "ES PARTE DE". Las propiedades que ha de tener –y que servirán para distinguirla de una asociación común– serán la **transitiva** ($xRy \wedge yRz \rightarrow xRz$) y la **asimétrica** ($(x,y) \in R \rightarrow (y,x) \notin R$).
Ejemplo:

Un elemento puede pertenecer, por supuesto, simultáneamente a varios agregados. Así, un párrafo puede estar agregado en varios documentos, y lo mismo ocurre con la frase con respecto al párrafo. Es, por tanto, asumible una cardinalidad "varios" en el término agregado –además de, por supuesto, en el agregando– de la asociación agregación.

Si tiene un comportamiento dinámico –un elemento es cambiado de agregado–, ésto no sería representable en un modelo estático como el modelo de objetos. Al menos en OMT; en UML, los modelos estáticos –vista estática, diagrama de clases– admiten un cierto dinamismo al incorporar conceptos de la clasificación dinámica como por ejemplo el estereotipo <<become>>, que podría usarse para modelar un comportamiento de este tipo.

En UML surge el concepto de **composición**, que intenta modelar una agregación con la semántica añadida de que las partes son *fijas*; es decir, los componentes están *permanentemente ligadas* al compuesto – son parte inamovible de él. Se dibuja con un rombo relleno.

Generalización o herencia.

En los orígenes del paradigma objetual en el campo de la programación, se pensó que la generalización era una panacea debido a las facilidades que en ella ofrecía para reutilizar código.
Después, con el traspaso del paradigma al análisis y diseño, surgieron las limitaciones. Sirva como ejemplo: "herencia de dos instancias de la misma clase" –algo completamente imposible en los entornos O.O actuales.

La generalización es una **proyección** de una instancia de una clase para poder manejarla desde el punto de vista de otras clases distintas, que representarán *escalones* el modelo; y poder con ello ejecutar métodos más

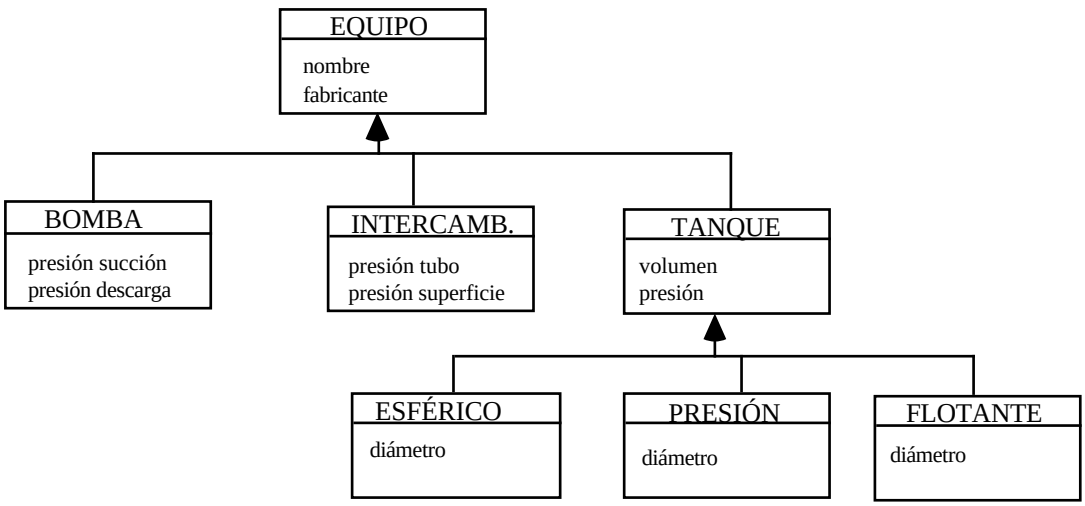
nítidos semánticamente en el escalón desde donde se está trabajando con dicha instancia. Ejemplo relacionado con la práctica II: los roles "arquitecto", "jefe de obra", etc. se proyectan como "usuario" (superclase) para manejar la autenticación en el sistema pero se usan desde sus respectivos roles a la hora de ofrecer el perfil de funcionalidad.

El modo de definir una generalización al modelar es el siguiente: si se observan atributos y operaciones comunes a un grupo de clases –las *subclases*–, se subirán éstos a una *superclase* y de ella *heredarán* todas las dichas clases.

La generalización intenta capturar una asociación con semántica "ES UN" ("ES UNA"). Pero esta expresión es confusa y puede llevar a errores de modelado –más correcto sería hablar de "SE PROYECTA EN", "TRABAJA", "FUNCIONA EN", etc.–.

Una instancia de una subclase es funcionalmente una instancia de todas sus clases ancestras. Los métodos o atributos son distintos según el nivel en el que se esté proyectando (algo que se ha perfilado ya al comienzo de este tema), con independencia de que se navegue de abajo a arriba o de arriba a abajo. Esto contrasta con el *overriding* ("superposición") implementado en ciertos lenguajes de programación.

Como apunte de notación, se representa con un triángulo por cada generalización. Tres puntos (…) indican una enumeración incompleta en la misma.



(Ejemplo de jerarquía de herencia – transparencia 238).

En la jerarquía de ejemplo se visualizan aspectos interesantes. En primer lugar, en un primer nivel se enumeran ***bomba***, ***intercambiador*** y ***tanque*** como posibles tipos de equipamiento. Puesto que se observa que comparten una serie de propiedades con la misma semántica en todos ellos –en particular, *nombre*, *fabricante* y posiblemente otros–, éstas se suben a un superclase ***equipamiento*** de que heredan todas.
Este concepto de la "misma semántica en todos ellos" es muy importante. Así, se observa en el ejemplo que *bomba* e *intercambiador* tienen unos atributos "presión succión", "presión tubo", etc. ¿Por qué no factorizar un atributo *presión* a la superclase?. La pregunta que habría que hacerse es; a pesar del nombre similar de estos atributos, ¿realmente comporten una semántica común que posibilite su manejo desde una instancia superior en la jerarquía? La respuesta es negativa, puesto que realmente estas presiones son completamente diferentes todas ellas (no tiene nada que ver "presión de tubo" con "presión de succión", "presión de superficie" o la "presión" de un *tanque*). Son atributos justificadamente distintos e independientes todos ellos.

En un segundo nivel, es posible refinar la jerarquía y matizar las características de un *tanque* según sea *esférico*, *de presión* o de *techo flotante*. En este caso se observa que el atributo *presión* sí tiene la misma semántica en toda ellas y por tanto se mantiene en la superclase; no ocurre lo mismo con *diámetro*.

La herencia ha sido en la POO sinónimo de reutilizar código. Hay una discrepancia, por tanto, bastante grande entre la programación y el análisis-diseño. El hecho de "copiar y pegar" una clase para adjuntarle comportamientos singulares o especiales es delicado desde el punto de vista global de un sistema.
La verdadera importancia de la herencia es la simplificación conceptual que supone.

Se observará que se emplean distintos términos para hablar de este concepto –*herencia*, *generalización*, *especialización* principalmente. Estos términos tienen no obstante un significado preciso:

Generalización.	**Derivación de similitudes (*de abajo a arriba*).**
Herencia.	**Forma o mecanismo por el que se comparten atributos y métodos en una generalización o especialización.**
Especialización.	**Discriminación de singularidades (*de arriba a abajo*).**

Así, una subclase especializa una superclase y una superclase generaliza una subclase.

La clasificación debiera de ser dinámica. Un objeto puede *perder* las singularidades que llevan a clasificarlo dentro de una subclase, con lo cual los vínculos a que accede desde ella dejarán de tener sentido. Debería, por tanto, de reclasificarse a otro nivel (por ejemplo perder la instancia de la subclase y pasar sencillamente a estar instanciado en la superclase).

Una instancia de una clase es funcionalmente una instancia de todos los ancestros de la misma. Todas las características de una clase ancestra tienen que ser aplicables a sus herederos; **no se pueden forzar o anular atributos**.

Superposición (*overriding*).

Según se ha comentado ya varias veces, en principio, los métodos o atributos son distintos según el nivel en el que se esté proyectando la jerarquía de herencia. No obstante, muchos lenguajes de programación orientados a objetos permiten un mecanismo de superposición u *overriding*.

Ésta es una característica que puede resultar peligrosa si no se maneja con cuidado – ya que, por ejemplo, se estaría ejecutando un método que no es el adecuado para el nivel desde donde se está viendo instanciado el objeto de acuerdo con sus relaciones con otros objetos. En caso de usarlo, es extremadamente importante **preservar la semántica** de la operación –de forma similar a como se ha comentado para el polimorfismo, y su uso ha de estar especialmente justificado – como por ejemplo, mejorar el rendimiento.

Módulos en OMT (paquetes en UML).

Un ***módulo*** o ***paquete*** es una form de agrupar la lógica del sistema desde distintos puntos de vista – por ejemplo: *funciones*, *despliegue*,

responsabilidades. Las clases se pueden referenciar en distintos módulos según su *visibilidad*.

Se comentará más sobre los paquetes en el tema de UML.

Modelización avanzada.

Agregación vs. asociación.

Como se ha dicho, la agregación presenta las siguientes características:

- La agregación es una forma especial de asociación que intenta capturar una semántica "es parte de".
- Los términos de una **agregación** son independientes del agregado.

Para discernir una agregación pueden utilizarse los siguientes criterios:

1. **Propagación** y **delegación**. Es natural que una operación sobre el agregado sea *propagada* a sus agregandos – o bien que el agregado *delegue* en ellos el resultado de la misma.
2. Relación necesariamente con la propiedad **asimétrica**.
3. Propagación de **atributos**.

Se debe, además, tener en cuenta las siguientes características que presenta una agregación.

Explosión y expansión.

- **Explosión**: un objeto se fragmenta en *partes* que pueden ser independientes del agregado.
- **Expansión:** los agregandos *complementan*, *expanden* o "*adornan*" el agregado.

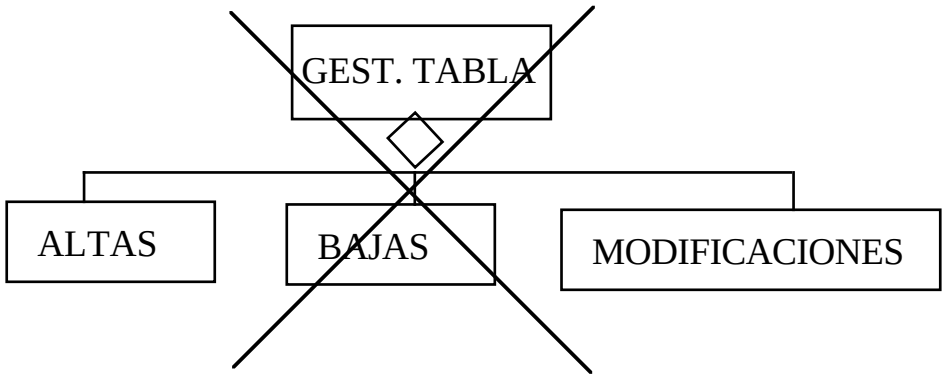
Los objetos agregandos adquieren una identidad conjunto con el agregado.
Ejemplo: los empleados NO SON agregados a la empresa.

En la práctica, la no distinción entre agregación y un asociación normal no causa problemas (es simplemente un término del lenguaje que añade riqueza semántica).

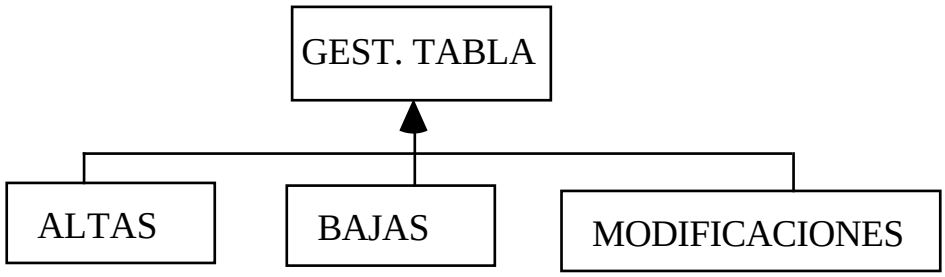
Agregación vs. generalización.

- La **agregación** relaciona instancias reales (varios objetos distintos) mientras que en la **generalización** sólo hay una identidad – relaciona clases – que simplemente se proyecta a distintos niveles.

Ejemplo:



(Modelo incorrecto)

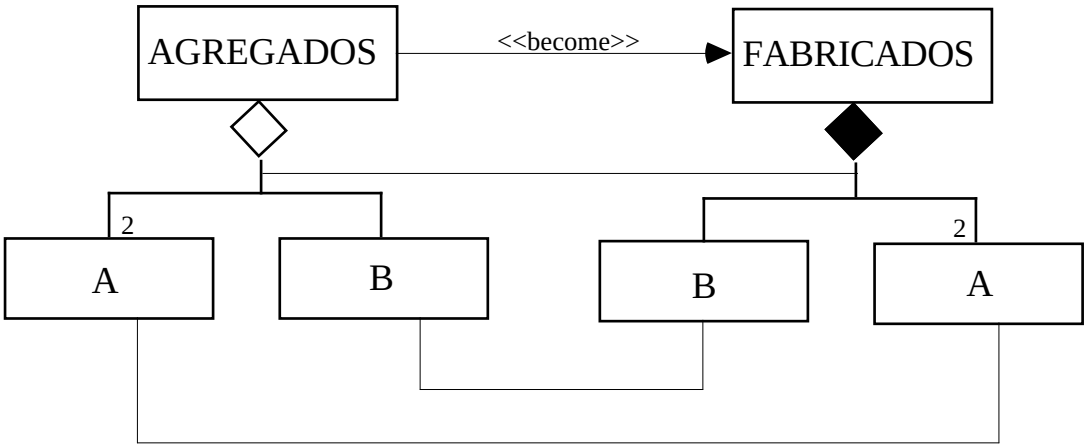


(Modelo correcto)

Se observa que *altas*, *bajas* y *modificaciones* **no pueden modelarse** como distintos objetos con distinta identidad – **no son objetos distintos** que estén agregados en la *gestión de tabla*.

- La confusión entre agregación y generalización surge porque en ambas se puede aplicar la *propiedad transitiva* (y la *asimétrica)*.
- La agregación permite manejar una conjunción de objetos mientras que en la agregación se trata de una disyunción. Ver ejemplo de la gestión de tabla.

Agregación vs. composición.



Se tiene un modelo inicial, dinámico, formado por explosión. Esto puede modelar, por ejemplo, un esquema o plano para la fabricación. Cuando esta agregación dinámica se fija –por ejemplo, porque se fabrica el producto a partir del plano– el resultante pasa a ser una *composición*, puesto que los componentes pasan a estar permanentemente ligados al compuesto – se cierra a algo inamovible.
En el ejemplo se modela esto con el estereotipo <<become>> de UML; no se puede modelar en OMT.
En el ejemplo, el link *regula* el proceso de que un objeto de la clase Agregados (en proceso de fabricación) pase a formar parte de la clase Fabricados. Evidentemente, se requiere para ello **clasificación dinámica**.

Agregación variable.

Agregación recursiva.

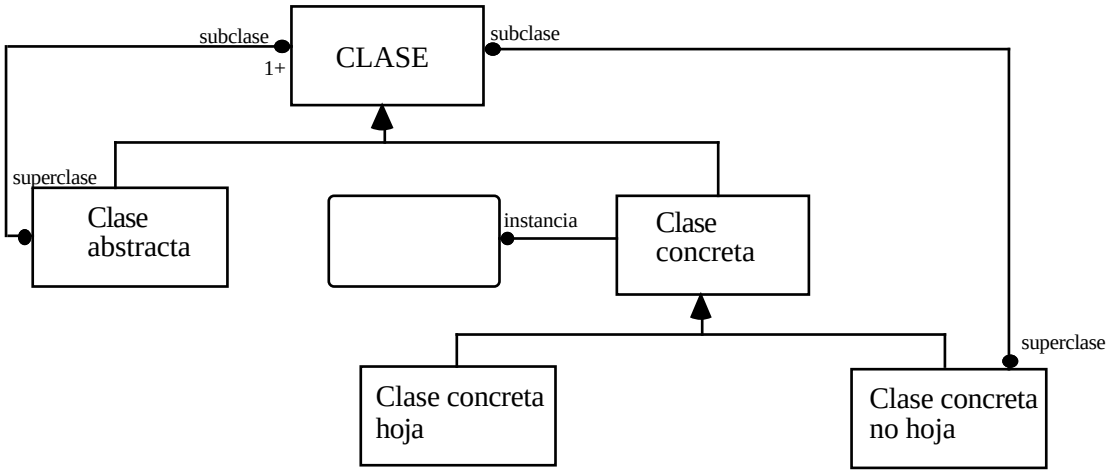
Propagación de operaciones.

La *propagación de operaciones* consiste en la propagación de un mensaje a una red de objetos. El caso más claro es la propagación de un agregado a sus partes. Es, pues, un indicador de la existencia de una agregación.
Si la operación en cuestión es polimórfica, es posible que se ejecuten distintos métodos en la propagación.

Clases abstractas.

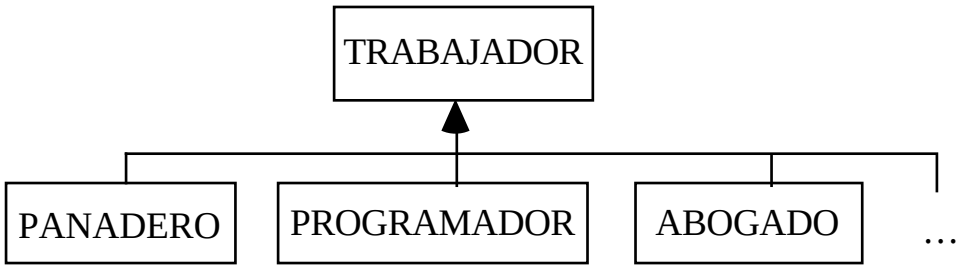
Una *clase abstracta* es aquella que no tiene directamente instancias, pero alguno de sus descendientes directos en la jerarquía de herencia sí puede tenerlas.

El siguiente metadiagrama de clases ilustra las situaciones de herencia que se pueden producir con clases concretas y abstractas.



Como se observa, una clase abstracta no puede ser hoja del árbol de herencia.
También se observa que el diagrama contempla la clasificación múltiple.

Sea la siguiente jerarquía de clases:

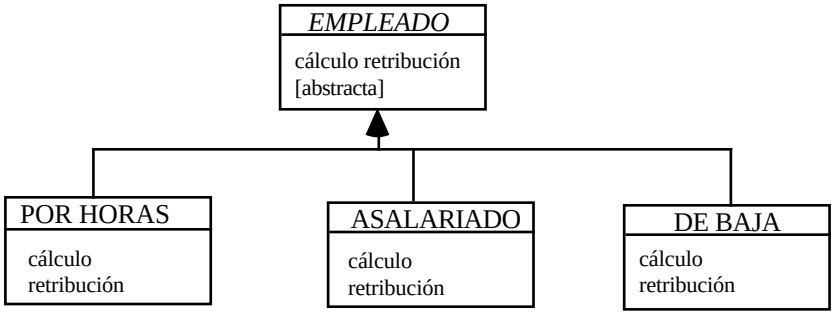


Como se ha mencionado, los puntos suspensivos (…) indican que no se han enumerado todas las posibilidades, pero pueden existir perfectamente en el sistema, aunque aparecerán simplemente como instancias de la superclase.
Por ello, la clase TRABAJADOR **no será abstracta** sino concreta.
En UML se puede –no así en OMT– modelar una clasificación dinámica y una reclasificación dinámica – un objeto instanciado en la superclase puede "pasar" a una nueva subclase (p.ej. CARPINTERO).

En ocasiones puede resultar útil crear una superclase abstracta cuando se observa que dos clases tienen los mísmos vínculos en la *sociedad de objetos* aunque sus atributos (o métodos) sean completamente diferentes.
Otras veces las clases abstractas aparecen de forma natural en el dominio de la aplicación. Tal es el caso de las **interfaces**.

Operaciones abstractas.

Una *operación abstracta* es aquélla definida en una clase abstracta pero que no contiene métodos. Éstos habrán, pues, de definirse en las subclases que hereden de la clase abstracta, que se superpondrán –*overriding*– al método nulo especificado en la misma.
Se ha comentado ya varias veces que, por razones de coherencia, simplicidad o seguridad en el modelado, los métodos asociados a una instancia deberían ser los propios del nivel de la jerarquía de herencia desde donde se está proyectando ésta – y por tanto desde cuyas relaciones con otros objetos se está manejando la instancia. Se ha dicho también que, si se viola este principio; es decir, si se admite la superposición u *overriding*, ha de ser por motivos debidamente justificados.
Uno de estos "motivos" –y de hecho **la única razón** que justificaría, en análisis y diseño, esta superposición– es en el caso de las **interfaces**.
Sirva el ejemplo siguiente:



> Sólo si la clasificación es disjunta es justificable la definición de una operación abstracta en la superclase. Es fácil de comprender por qué: si se admite, por ejemplo, un empleado clasificado en POR HORAS y en DE BAJA, ¿qué ocurre con la operación "cálculo retribución" cuando la instancia se proyecta en la superclase? ¿qué método se debería ejecutar – el proporcionado por "POR HORAS" o el de "DE BAJA"?

Se denomina **protocolo** a la orientación semántica que tiene una característica definida en una superclase abstracta – notablemente una operación abstracta. Al igual que ocurría con el polimorfismo, los entornos de programación limitan el concepto de "orientación semántica" a que tengan el mismo número de argumentos y cada uno con el mismo tipo – tanto de entrada como de salida.

Generalización como extensión y restricción.

Una instancia de una clase es funcionalmente una instancia de todos los ancestros de la misma. Todas las características de una clase ancestra tienen que ser aplicables a sus herederos; no se pueden forzar o anular atributos.

Herencia múltiple.

La *herencia múltiple* es un mecanismo que permite a una clase tener más de una superclase y heredar todas las características de todas ellas. La herencia múltiple resulta fundamental para acercar el modelado a la forma de pensar del mundo real –un objetivo de la orientación a objetos– y solucionar los problemas que la herencia simple tiene para representar ciertas situaciones, aunque tiene como contrapartida una mayor complicación con respecto a la herencia simple (que implicaba siempre jerarquías arborescentes).

Una característica que pueda ser heredada por más de un camino se adquiere una sóla vez. El problema es que puede haber ambiguedades entre ambas rutas de herencia. OMT no propone una solución a este problema; se remotará, pues, en el tema de UML.

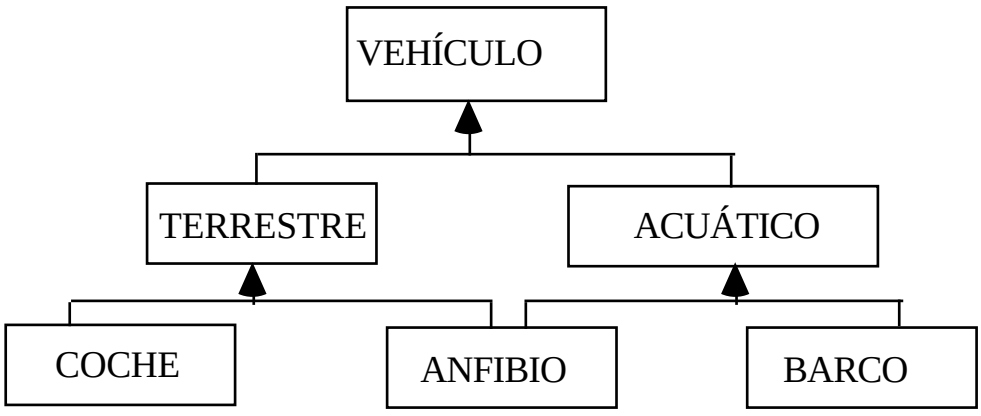
Tipologías.

Para hablar de herencia múltiple es fundamental apreciar las distintas situaciones que se pueden producir mediante el uso de la misma. En particular, se puede considerar una especialización como *disjunta* o *solapada*. La notación será un triángulo vacío o relleno:

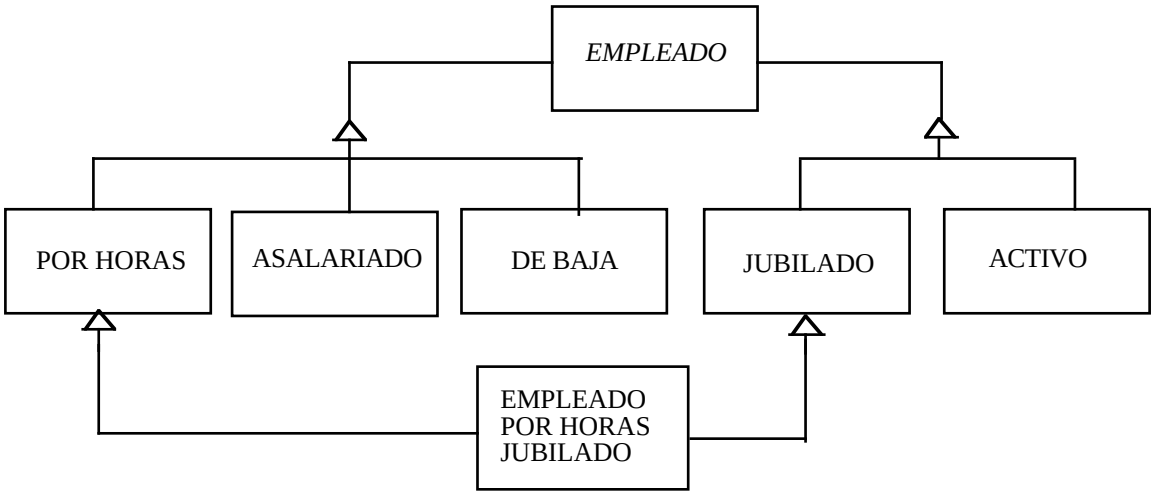

Herencia solapada.


Herencia disjunta.

En el siguiente ejemplo, TERRESTRE ó ACUÁTICO es una especialización *solapada* de la superclase VEHÍCULO; y ANFIBIO es la subclase donde se solapan. Se trata, pues, de una ***herencia múltiple solapada***.

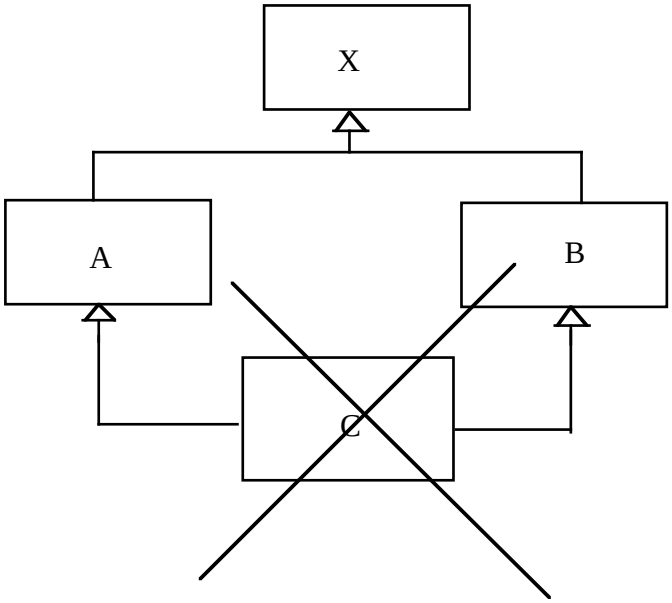


En contraste, en el siguiente ejemplo, las categorías POR HORAS, ASALARIADO y DE BAJA así como JUBILADO y ACTIVO son *disjuntas*. La subclase "empleado por horas jubilado" hereda tanto de POR HORAS como de JUBILADO. No puede considerarse una "herencia múltiple solapada" pues, aunque reúne las categorías de "por horas" y "jubilado", éstas no forman parte de una misma rama de herencia solapada sino que son distintas polas de una herencia disjunta.



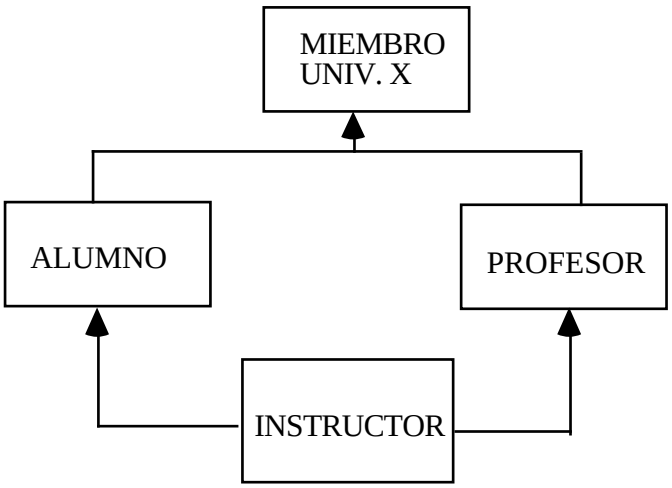
De todas formas, la manera más correcta de modelar ésto es mediante clasificación múltiple –donde una instancia puede estar clasificada en varias clases simultáneamente; por ejemplo en "por horas" y en "jubilado"–. De lo contrario, el número de posibles subclases crecerá combinatoriamente y el sistema no será muy flexible.
Por ejemplo, un empleado podría trabajar "por horas" en el turno de mañana y "asalariado" en el turno de tarde, pero el anterior modelo no lo permitiría. Si se decidiera resolver este problema mediante herencia múltiple, habría que crear una gran cantidad de clasificadores en lugar de estos tres (en concreto, se necesitarían ocho clases para todas las posibles combinaciones – y ello sin contar con la rama de "jubilado" o "activo"). Por el contrario, si el entorno admitiera la clasificación múltiple, bastaría con cambiar la herencia para que fuera *solapada* en lugar de *disjunta*.
Se retomará este tema en el apartado *Los problemas de la herencia. Workarounds*.
En resumen, como principio de modelado, los modelos deben de ser flexibles.

Como apunte adicional, es evidente que una clase no puede heredar simultáneamente de dos clasificadores de una herencia disjunta (sería una contradicción).



Herencia múltiple accidental.

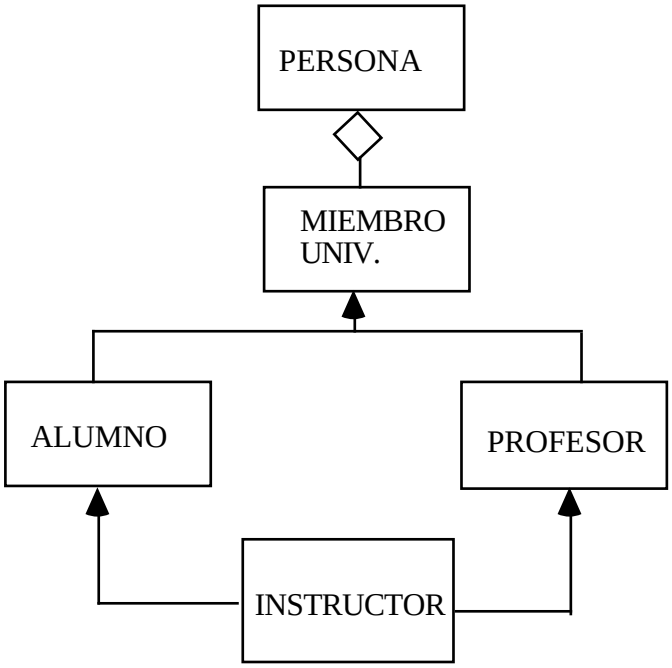
Pártase de la siguiente situación, sacada del dominio de los miembros de una universidad X: "*un instructor es un alumno que además, debido a que ha cumplido ciertos requisitos, es además profesor de alguna asignatura*".



Bien; considérese ahora que un alumno puede optar a ser instructor en *otra* universidad *distinta a donde es alumno*. Por ejemplo, es alumno en la universidad X con la calidad de instructor y, debido a sus derechos como tal, ejerce de profesor instructor en la universidad Y. El primer problema que surge es que la clase "miembro univ. X" se refiere a una universidad X; si se tienen en cuenta las dos universidades, el objeto perdería su identidad ya que

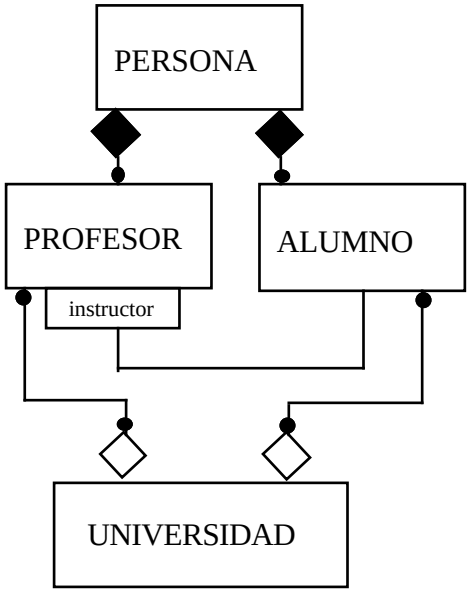
habría, por un lado, un "miembro univ. X" y por otro lado un "miembro univ. Y".

Podría pensarse en desligar la identidad de la pertenencia a la universidad mediante una agregación:



(Se entiende aquí que la clase "MIEMBRO UNIV." está referida a una universidad).
Con este segundo modelo, una instancia de "persona" podría agregar dos instancias de "miembro univ."; una referida a la universidad X y otra a la universidad Y. La instancia de la universidad X será además instancia de "alumno" y de "instructor", y la de la universidad Y será de "profesor" e "instructor". Pero el problema es que en la herencia, una instancia puede proyectarse como una instancia de todos sus ancestros. En particular, la instancia de "instructor" de la universidad X podrá proyectarse como "profesor" de la misma, lo que no tiene sentido; y lo mismo sucede con "instructor" y "alumno" en la universidad Y, lo que no tiene sentido.

En realidad, **no existe una solución dentro de la herencia** para este problema.
En su lugar, deben emplearse rodeos o *workarounds* para modelar el problema. Uno de los más satisfactorios consiste en descartar la herencia y utilizar, en su lugar, *agregación de roles*. Por ejemplo, el siguiente modelo resuelve con éxito la casuística anterior:

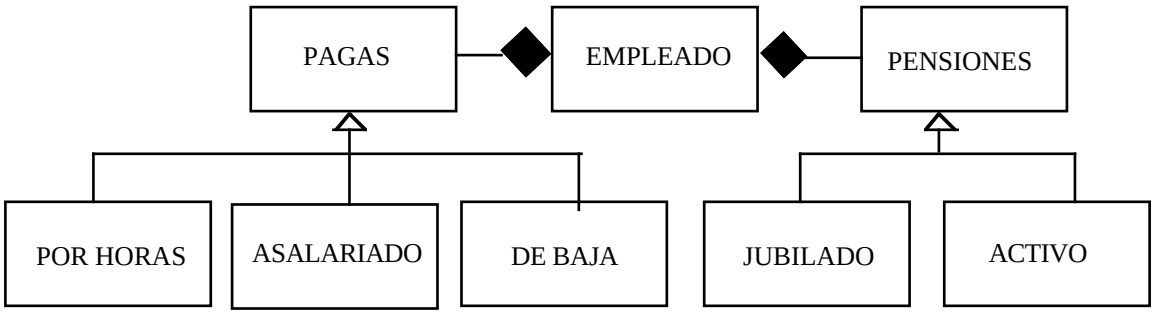


Los problemas de la herencia. Workarounds.

Se ha visto en el apartado anterior que la herencia es problemática y puede causar errores sutiles en el modelado. En muchas ocasiones es, pues, conveniente descartarla y emplear en su lugar uno de los siguientes rodeos o *workarounds*.

- Descartar la herencia y usar en su lugar agregación de roles (delegación).
- Heredar de la clase más importante y agregar en el resto.
- En lugar de utilizar *herencia múltiple solapada* a varios niveles, factorizar todas las generalizaciones en el mismo nivel.

Delegar empleando agregación de roles.



- Es el tipo de modelo más flexible que se puede construir, puesto que admite todo tipo de casuísticas. En particular, resuelve el problema, que se comentaba en el apartado de la herencia múltiple, del empleado que trabaja por horas por la mañana y como asalariado por la tarde.

- Como contrapartida, no hay una "herencia automática" en las operaciones sino que ésta debe implementarse manualmente: la operación que se ejecuta en el agregado **ha de delegarse** en el agregando o los agregandos correspondientes.

Heredar de la clase importante y delegar en el resto.
Factorizar generalizaciones.

Regulación.

Un modelo debe ser preciso; así pues, el uso de mecanismos de regulación en un modelo es un indicador de la calidad del mismo.

Imperativos, restricciones y condiciones.

La *multiplicidad* en si misma es una forma de restricción. Pero el lenguaje admite la especificación de cualquier tipo de *restricciones*. Se colocan entre los símbolos de corchetes [y] en UML, o bien { y } en OMT. Pueden ser:

- restricción de asociación.
- restricción de clase.

Pueden estar escritas.

- En un lenguaje formal. En UML, se define el lenguaje OCL con este propósito.
- Con notación matemática.
- En lenguaje natural.

Se retomará este tema cuando se hable de UML.

Objeto derivado.

En orientación a objetos puede utilizarse un concepto de derivación que es más potente que los meros "atributos derivados" utilizados en diseño de base de datos. Puede derivarse cualquier objeto, por ejemplo:

- una asociación.
- un objeto.

La notación es el símbolo /.
Puede derivarse en **cascada**.

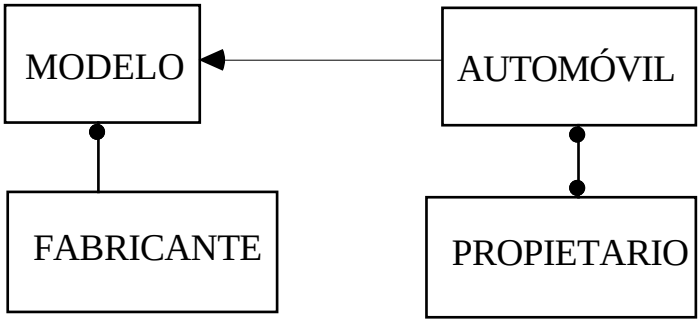
Metadatos.

- *Metadatos en el modelado*: por ejemplo; una clase describe un conjunto de instancias. Es un *metadato* de las mismas. Un patrón describe una serie de modelos. Las clases pueden ser consideradas **objetos**

("metaobjetos") y tienen asociadas operaciones; la más común de ellas es la creación de nuevas instancias. La notación empleada es el símbolo $. Las clases que describen otras clases se llaman *metaclases*.

- *Metadatos en la realidad*: por ejemplo planos de un proyecto, dibujo de un ingeniero, catálogo de partes.

Si en un modelo existen metadatos, éstos serán una ayuda importante para la regulación.

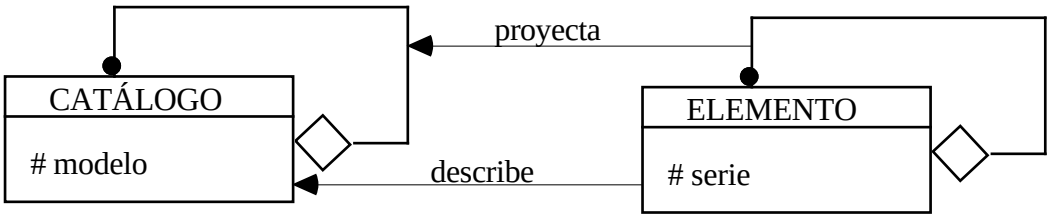


(Ejemplo: un "modelo" es un "metadato" de automóvil. Ojo: aquí no hay *homomorfismo*, un concepto que se introducirá a continuación, puesto que el homomorfismo se centra en los vínculos y no meramente en los objetos. Para representar ésta situación es suficiente con la línea discontinua cuya semántica es "es una instancia de").

Homomorfismos.

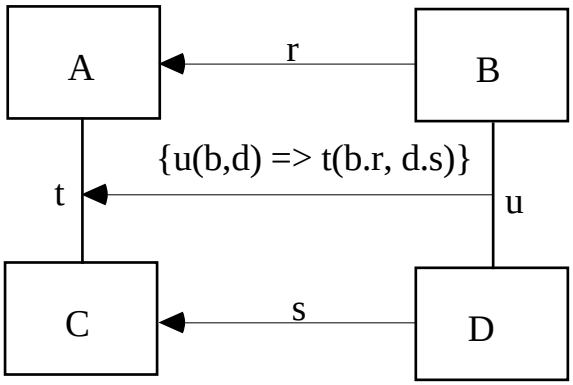
Es el concepto más avanzado de que se dispone para regular. Se trata de una **proyección** entre dos asociaciones, donde una de ellas *regula* la otra. Es un concepto extremadamente útil que aparece con frecuencia en la realidad.

Ejemplo muy sencillo:



No puede existir una descomposición física de un "elemento" en sus partes (que son a su vez elementos) si no está descrita tal descomposición en el "catálogo". Es decir, la relación entre los ítems del "catálogo" y la relación entre los "elementos" son **homomórficas**.

La forma general es la siguiente:



La asociación entre A y C **regula** el que pueda existir una asociación entre B y D. Es decir, además de haber una relación de instanciación entre objetos, también la hay entre sus asociaciones. Es un concepto muy versátil.

El lenguaje unificado de modelado (UML).

Introducción.

UML es un lenguaje libre en su expresión (es decir; las vistas, diagramas y símbolos son libres) donde sus términos están acotados semánticamente. No obstante, propone una serie de vistas, diagramas y símbolos estándar.

Un **modelo** es una herramienta en ingeniería que permite calcular y probar.

- Describe cómo es o cómo se quiere que sea el sistema y sus comportamientos.
- Proporciona plantillas para verificar la construcción del sistema.
- Y por supuesto, documenta las decisiones de análisis y diseño.

Un modelo se realiza desde el contexto de un ***nivel de significación***. Omitirá aquellos elementos que no son relevantes en su nivel de significación. Es una regla de modelado importante el hecho de **no incorporar todo lo que sea irrelevante en el contexto**. Es una **simplificación de la realidad**.

Hay varios enfoques para el modelado:

- *Enfoque de BB.DD*: Modelos E/R. Procedimientos almacenados. Disparadores (triggers).
- *Enfoque estructurado*: DFDs y otros diagramas.

- *Enfoque orientado a objetos*. Universo de clases e interacciones. Tiene la ventaja de que se utiliza la misma notación (por ejemplo en UML) en todas las fases del desarrollo (análisis, diseño, …).

Los mejores modelos son los que se mantienen ligados y se adaptan a la realidad.

Un único modelo no es suficiente. En todo proyecto de ingeniería se necesitará plantear varias vistas; las propuestas por UML son estándar y cubren muy diversos aspectos, pero como se ha dicho, el lenguaje es libre en su expresión.

Con respecto al desarrollo software, existen varias orientaciones. Por ejemplo:
- Perspectiva algorítmica (*divide y vencerás*).
- Orientación a objetos.

La aplicación de UML se centra en este último, la **orientación a objetos**. Puede utilizarse, entre otras cosas, para:
- *Visualizar*, *especificar*, *construir* y *documentar* sistemas O.O.
- Modelar reglas de negocio. Las reglas de negocio –véase definición de "racionalización" en el primer tema– incorporan **toma de decisiones** sobre cómo hacer las cosas y son independientes del software. Son ejemplos de reglas de negocio las siguientes:
  - Cómo se sincronizan líneas de producción.
  - Cómo se formatea un documento usado para una relación con terceras partes.

Lo importante en UML es la **semántica**, siendo irrelevane lo *visual* –excepto para ciertas herramientas que procesan modelos–. Es decir, los elementos derivan su significado de la definición semántica que se documente en UML – aunque no totalmente, pues es cierto que hay información sutil en la forma de dibujar. Por tanto, y como se ha dicho, pueden definirse nuevos símbolos, nuevas vistas, nuevos diagramas, etc.; siempre y cuando se documente el significado de todos ellos usando los términos de UML.

- **Abstracción frente a detalle**. A mayor nivel de abstracción, menor detalle y viceversa.

- **Especificación e implementación**. Es evidente que es necesario detallar a fondo el *qué* antes de considerar el *cómo*.
- **Descripción frente a instancia**. El concepto de *metadato* que ya se ha perfilado en el tema anterior.

Descomposición de los modelos UML.
Áreas: estructural, dinámica, gestión, externa, física.

Vistas.
Vista estática.
Diagrama de clases.
Vista de casos de uso.
Diagrama de casos de uso.
Vista de implementación.
Diagrama de componentes.
Vista de despliegue.
Diagrama de despliegue.
Vista de la máquina de estados.
Diagrama de estados.
Vista de actividades.
Diagrama de actividad.
Vista de interacción.
Diagrama de secuencia.
Diagrama de colaboración.
Vista de gestión.
Diagrama de paquetes.

Vista estática.

Algunos autores la consideran la "base" de UML; no obstante, en un proceso correcto de análisis y diseño, el órden de realización de los diagramas sería más bien primero *casos de uso*, después *clases colaboradoras* y después *vista estática*.

La **vista estática**, con su *diagrama de clases*, tiene que soportar un comportamiento dinámico basado en la **colaboración**. Ha de capturar la estuctura del sistema y "objetualizarla". Por supuesto, como se ha dilucidado en el tema de la O.O., a estos objetos tiene que poder otorgárseles una

identidad válida. Se usan para ello elementos de modelado discretos llamados *clasificadores*.
Un **clasificador** tiene identidad, posee relaciones, envía mensajes y posee un comportamiento sensible al contexto. Se observa que todas estas propiedades son las propias de los objetos –según se ha visto en el tema de la orientación a objetos–; pero el concepto de clasificador UML admite, además de "clases", otros componentes.

- Un *actor*.
- Una *clase*.
- Una *clase en un estado*. De esta manera el modelo queda mucho más legible, especificando solamente las relaciones adecuadas para el estado en el que esté la clase, en lugar de colocarlas todas bajo el símbolo de clase general cuando muchas de ellas no serán válidas.
  Quizás, en un nivel más bajo de realización semántica, ésta se convierta en una clase que especialice la clase original.
- Un *rol*.
- Un *componente software* dentro del que no se desea profundizar más para analizar su estructura objetual.
- Una *interfaz*.
- Un *nodo*, de forma similar al componente software.
- Una *señal* (estereotipo <<signal>>). Es un *mensaje* que se modela como una clase, puesto que puede persistir en el tiempo.
- Un *paquete* o *subsistema* dentro del que no se desea profundizar más para analizar su estructura objetual..
- Un *caso de uso*. Este es quizá el elemento más sutil de los que se han citado, pues no está claro que cumpla los requisitos mencionados. Se admitirá **sólo** cuando el modelo está en desarrollo y el caso de uso no está concretado con clases colaboradoras.

Como se ha mencionado, la estructura estática diseñada ha de cumplir el *principio de ocultación* para el nivel de significación que se trate. En este caso, que algo sea significativo en el sistema implica que proporciona o presta servicios.

Entre los *clasificadores* pueden dibujarse las siguientes relaciones:

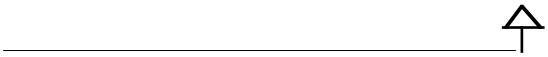
Asociación.

Dependencia. 

Flujo: caso especial de dependencia (<<become>> etc.).

Uso: otro caso especial de dependencia.

Generalización. 

Realización. 

Especialización de interfaces.

Niveles de significación.

Un *nivel de significación* especifica el nivel de detalle o abstracción que se necesita en un contexto dado. Por ejemplo: *análisis*, *diseño* e *implementación*.

De esta forma, los conceptos de implementación no relevantes en el mundo real se ocultan en el nivel de *análisis* (*principio de ocultación*). La implementación *realiza* las clases diseñadas con otras más lógicas.

Dependencias.

Entre los tipos de dependencias cabe citar: flujo, instanciación, realización (ya se han mencionado), importación, amigo (C++), derivación, llamada, envío, uso y traza (la relación más débil de todas).

Flujo.

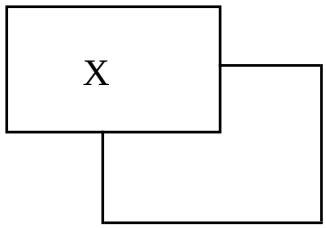
Son relaciones entre distintas formas del objeto. Por ejemplo los estereotipos <<copy>> y <<become>>.

Realización.

En un nivel de significación de más bajo nivel se *realiza* la descripción general con versiones más detalladas (añadiendo características respetando su semántica). Una realización al mismo nivel de significación puede ser considerada una *herencia* – o bien el uso de *interfaces*.

Diferencias notacionales con OMT.

Asociaciones monoinstancia.



En UML, la anterior notación indicará que *dos instancias distintas* se asocian. Puede relacionarse consigo mismo si la clase aparece dos veces. Son ejemplos la herencia, el uso de estereotipos <<copy>> y <<become>>, roles, estados…

Agregación y composición.

Ya se han explicado con mucho detalle en el tema de la orientación a objetos.

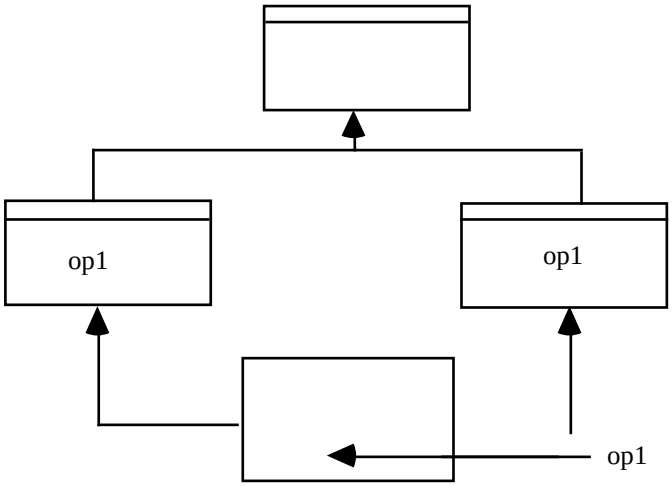
Direcciones.

UML admite bidireccionalidad en las asociaciones, especificando una *navegabilidad*.

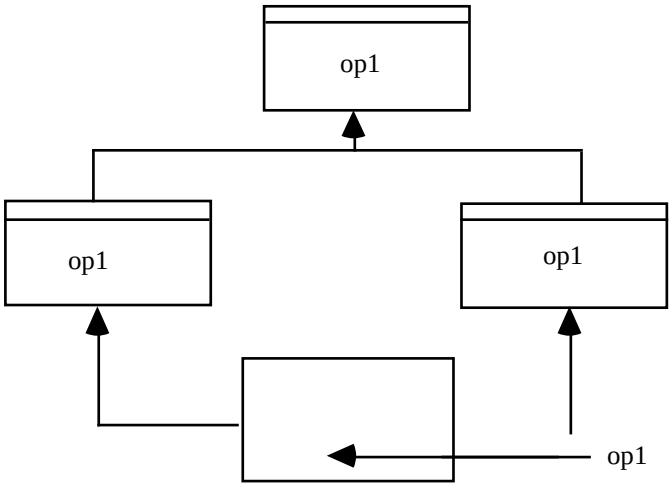
Herencia.

En la **herencia simple**, si un atributo está declarado en un antecesor, no puede ser declarado también en un descendiente. De lo contrario, se considerará que el modelo está mal formado. Una operación se puede redeclarar, pero si son consistentes semánticamente (véase el principio de modelado que se comentaba en el tema de orientación a objetos).

Se analizará ahora la **herencia múltiple**. Con respecto a los atributos, la situación es la misma (no se pueden redeclarar). ¿Qué ocurre al redefinir una operación común a varios ancestros?



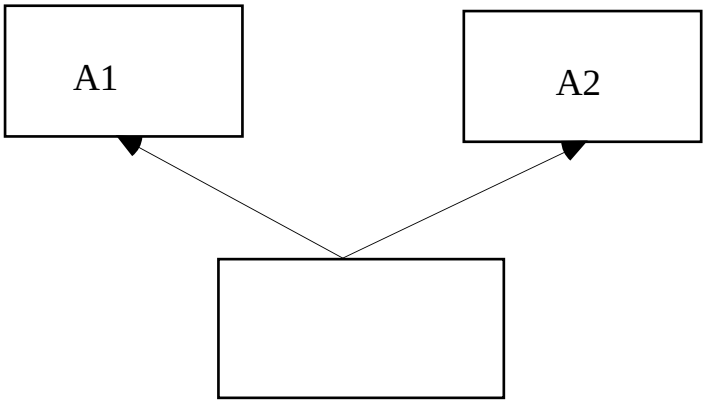
MODELO MAL FORMADO: *op1* no tiene una referencia común. UML exige que el atributo común a varios ancestros tenga una referencia común. Esto puede interpretarse como que tiene que haber asociaciones comunes que validen la herencia de este atributo. La declaración es conflictiva.



MODELO BIEN FORMADO. Aún así, UML no especifica una regla para deshacer el conflicto en la herencia de *op1* (hay tres posibilidades).

Clasificación simple y múltiple.

No hay ninguna necesidad lógica de que un objeto pertenezca o se instancia en una sóla clase.



Muchos entornos de programación no lo toleran. Los artefactos de implementación o *workarounds* pueden ser los siguientes:

- Usar **herencia múltiple** si está permitida, con todos los problemas que se han visto al hablar de ella.
- Emplear los *workarounds* para la herencia múltiple estudiados en el apartado *Los problemas de la herencia. Workarounds* del tema de orientación a objetos.

Clasificación estática y dinámica.

La *clasificación dinámica* permite a un objeto camiar de clase. UML la admite –véase por ejemplo el estereotipo <<become>>–. Hay que tener en cuenta:

- Las propiedades perdidas no se recuperan.
- Las propiedades adquiridas se inicializan a su valor por defecto.

Combinación de clasificación múltiple y dinámica.

La combinación de clasificación múltiple y dinámica permite implementar el patrón de ***roles*** de forma más natural que lo que se ha visto en la "delegación empleando agregación de roles". Un objeto puede ganar o perder clases –roles– a lo largo de su vida.

Vista de los casos de uso.

Un *caso de uso* es una *pieza de funcionalidad interactiva*. Se produce una secuencia de mensajes entre el *sistema* y los *actores*. Para poder modelar una organización de esta manera, es necesario desglosarla en funciones, procedimientos, responsabilidades… Es requisito indispensable que la organización esté **racionalizada**.

Un actor puede ser una *posición en la organización*, una *persona*, un *sistema*… en general es todo aquello que es **opaco** al sistema pero que interactúa y envía mensajes con éste.

Los actores se deben definir como jerarquías. Varios usuarios físicos pueden estar ligados con el mismo actor.

Un caso de uso **no revela** la estructura interna del sistema; ni tiene relación alguna con dicha estructura interna.

En la vista de los casos de uso no se indica la secuencia o dependencia entre ellos – esto se hará en la **vista de actividad**. Cada caso de uso se puede modelar con **clases colaboradoras**.

Estructura de los casos de uso.

- *Generalización*.
- *Inclusión*: incorpora otros casos de uso. El caso de uso base *necesita* la lógica base y la lógica incluida.
- *Extensión*: lógicas complementarias que se añaden a la lógica base y que pueden ser utilizadas si es oportuno, pero ello no es imprescindible para el funcionamiento normal. Por ejemplo una ayuda contextual.

Vista de la máquina de estados.

La vista de la máquina de estados modela un **comportamiento dinámico**. Un objeto recibe eventos y responde a ellos. Un evento modela un *suceso* del mundo real. Son eventos, por ejemplo:

- Llamadas (síncronas).
- Señales (asíncronas).
- Cambio en ciertos valores.
- El paso del tiempo.

Estados.

Un objeto estára en un **estado**, que lo hará o no *sensible* a ciertos eventos. Es decir: en el mismo *estado* se ejecuta la misma acción ante el mismo evento concreto. La acción podrá variar en estados diferentes.

Máquinas de estados.

Es un grafo de ***estados*** y ***transiciones***. Es una vista *reduccionista* del sistema, ya que separa un objeto y examina su comportamiento aislado. Para examinar el comportamiento conjunto es mejor utilizar la **vista de interacción**.

Clasificación y tipos de eventos.

- Señal (<<signal>>): es **asíncrona**. El objeto A continúa su hilo de ejecución. El objeto B puede atender la señal en el momento que se recibe, dejarla en una cola, o abrir un nuevo hilo para atenderla.
- Llamada (<<call>>): es **síncrona**. El objeto A queda a la espera de recibir la respuesta por parte del objeto B (<<return>>). No obstante bien puede abrir un hilo para ejecutar la llamada y continuar su ejecución en otro hilo.
- Eventos de cambio: es una forma declarativa de esperar hasta que una condición esté satisfecha. Puede implicar una computación continua, por lo que debe ser usada con cuidado. La diferencia con una condición de guarda es que en ésta sólo se evalúa el condicional antes de efectuar una transición.
- Eventos de tiempo.

Transiciones.

Pueden llevar asociada una acción.

- De entrada al estado.
- De salida del estado.

Condición de guarda.

Acciones.

Se pueden ligar a una transición o a un estado asociado con un evento.

Cambio de estado.

Completada la ejecución de la acción ligada a una transición, el estado destino de ésta pasa a ser el estado activo. Esto puede activar a su vez acciones de entrada y salida en los estados implicados.

Estados anidados o compuestos.

Pueden ser:

- Estado concurrente compuesto (varios hilos en paralelo). Se dibuja con una línea discontinua separando las dos bandas.
- Estado secuencial compuesto. Simplemente se agregan los subestados dentro de un estado grande.

Estados especiales.

- Estado inicial.
- Estado final.
- Unión.
- Historia.

Vista de actividades.

Es una forma especial de máquina de estados que implica a varios clasificadores. Modela computaciones y flujos de trabajo. Un ***estado de actividad*** representa distintos estadíos en la ejecución de una computación o un proceso en la organización. Es decir, está orientada a procesos y no a eventos externos.

Puede tener:

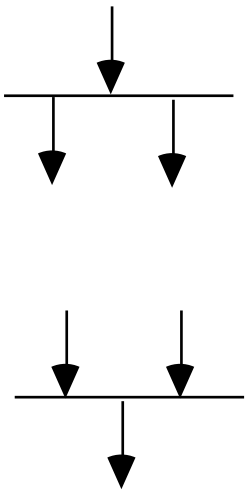
- hilos.
- divisiones de control (hilos concurrentes).
- bifurcaciones.
- sincronismos.

Conceptos de la vista de actividades.

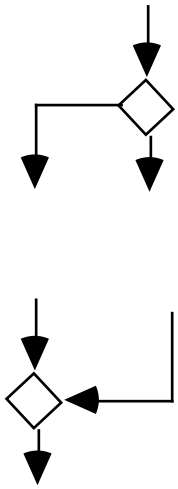
- *Calle*: recurso para organizar las actividades para modelar de forma muy cercana a la organización (por ejemplo: departamento de ventas, almacén…).
- *Sincronismo*: Unión y división de hilos.
- *Flujo de objetos*: Los objetos *fluyen* pasando por distintos estadíos de la computación y por distintas calles. Un objeto puede tener varias entradas (unión) y varias salidas (división).

Como se observa, es ésta una vista muy cercana a la organización. Tanto es así, que algunas organizaciones modelan sus procesos de negocio y flujos de trabajo mediane UML empleando, entre otras, esta vista.

Concurrencia vs. alternancia.



(a) AND (barra de sincronismo)



(b) OR